An Effective and Efficient Approach to Improve Visibility Over Network Communications

Marco Zuppelli[1*], Alessandro Carrega[2], and Matteo Repetto[1,3]
[1]National Research Council of Italy, Genova, Italy
{marco.zuppelli, matteo.repetto}@ge.imati.cnr.it
[2]National Inter-university Consortium for Telecommunications (CNIT), S2N Lab, Genoa, Italy
alessandro.carrega@cnit.it
[3]CNIT, IMATI RU
matteo.repetto@cnit.it

Received: June 25, 2021; Accepted: September 2, 2021; Published: December 31, 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.

Keywords: eBPF, network covert channels, network monitoring

1 Introduction

The software industry has probably never delivered so many innovations and so fast as in the last twenty years. Following the progressive introduction of virtualization paradigms first and cloud models later, computing architectures have progressively evolved from monolithic to distributed and scalable frameworks, up to the introduction of micro-services and service mesh patterns [1]. Motivated by the need for more agility in the development and operation pipelines, applications are today commonly decomposed in multiple business units, each one with specific functions. The large availability of public infrastructures and devices allow the deployment of such units in a pervasive and distributed fashion, bringing the opportunity to build ubiquitous, mobile, and immersive services, which were even unthinkable a decade ago [2].

The dark side of this evolution is the growing difficulty to monitor and inspect this new breed of applications and services when looking for malware, intrusions and other forms of passive or active

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 12(4):89-108, Dec. 2021
DOI:*10.22667/JOWUA.2021.12.31.089*

*Corresponding author: Institute for Applied Mathematics and Information Technologies, National Research Council of Italy, Via de Marini 6, Genova, Italy, I-16149, Web: `http://imati.cnr.it`

attacks. As a matter of fact, while more conventional applications commonly lie within a safe security perimeter (being it the traditional enterprise network or a more modern cloud infrastructure), distributed services can be deployed over a mix of cloud, edge, and personal devices connected through the global Internet [3]. This greatly increases the attack surface, because of the specific attack vectors for each environment. It also makes the deployment of legacy cyber-security appliances more difficult, because they often assume full visibility over the application and its execution environment (hardware, software, operating system, file system, network) but this is hindered by the same nature of cloud models [1]. Finally, being largely conceived as monolithic applications (at least from a functional perspective), they should be replicated for each set of business units in a different environment, hence leading to large inefficiency especially in case of lightweight containers (e.g., Docker) [4].

To cope with such an increasingly challenging scenario, in this paper we investigate the usage of the extended Berkeley Packet Filter (eBPF)[1] implemented in the Linux kernel for effective and efficient monitoring of distributed resources. Even if the original BPF framework was explicitly conceived for network packet filtering, the extended version widens the scope to more general kernel and application hooks, including *kprobes*, *uprobes*, *USDT* (Userland Statically Defined Tracing), *kernel tracepoints*, *lttng-ust* (Linux Trace Toolkit Next Generation Userspace Tracer), as well as additional network hooks such as XDP (the eXpress Data Path) and TC (the Traffic Control subsystem). Beyond the possibility to get notified when specific execution points are reached, the most useful feature of eBPF is the possibility to run custom code, which in practice dynamically extends the kernel behavior. Being fully integrated and available in all recent vanilla kernels, the eBPF represents the ideal monitoring and tracing mechanism for cloud native services and even IoT devices [5].

The range of potential use cases which can be addressed by eBPF is virtually unlimited, however in this paper we restrict our focus on an emerging class of threats that have been observed in many real-world attacks yet have remained undetected for long [6]. Specifically, we showcase how eBPF can be used to gather information useful to drive the detection of a generic class of covert channels, i.e., hidden communication paths implemented by hiding information within network packets. Even if different types of channels exist, we will focus on covert channels built by directly storing information within fields of the TCP/IP protocol headers [7].

The idea of using eBPF to spot the presence of anomalies or covert communications has been originally introduced in [8, 9], covering a limited number of cases for IPv6 traffic. Later, a packet inspection tool was engineered for this purpose [10]. In this work we extend the original approach by considering additional protocols, namely, IPv4, TCP and UDP. Moreover, we investigate its efficiency, in terms of resource usage, with respect to more traditional tools. Specifically, we consider alternative implementations for detecting covert channels built via: i) `libpcap`, which is one of the most widespread framework for packet processing outside of the kernel, and ii) extensions to Zeek (formerly Bro), the de-facto standard network analysis platform. Our comparison includes both the complexity of the implementation as well as a deep analysis of resource consumption, including CPU, memory and disk usage. Our work shows that eBPF programs lead to more flexibility than existing appliances in inspecting network headers, and with lower requirements on computing resources. However, since our approach takes advantages of Python classes, it results in less efficiency than pure C implementations, as in the case of libpcap.

The rest of the paper is organized as follows. Preliminary, we discuss the background behind our work and give an overview of related work in Sec. 2. Sec. 3 briefly describes our detection mechanism for spotting network covert channels in protocol headers. We discuss the difficulties to create efficient monitoring processes for covert channels with existing cyber-security appliances in Section 4, and then we describe alternative implementations of this mechanism in Sec. 5, where we consider the set of aforementioned technologies and tools. We provide a complexity analysis in Sec. 6, which gives a quick

[1] `https://prototype-kernel.readthedocs.io/en/latest/bpf/index.html#introduction`

understanding of the level of expertise and difficulty for further extending the proposed tools, whereas numerical results from the extended experimental evaluation are reported in Sec. 7. Finally, we give our conclusion and plans for future work in Sec. 8.

2 Background and Related Work

Originally introduced as a mechanism to enable the sharing of information among processes running within a single computing infrastructure (see for example the seminal work of Lampson [11]), nowadays covert channels are used to support different sophisticated attack schemes. For instance, channels can be used to allow processes or applications sealed in separate execution enclaves to leak sensitive data or to exchange signalling to gather information on the underlying hardware [12]. Yet, a recent trend exploits covert channels to implement cloaked communication paths within network conversations, i.e., the secret information is hidden within packets or encoded in traffic characteristic such as the throughput or the delay [13]. Thus, modern malware is often endowed with network covert channels to remain "under the radar" while retrieving additional payloads, orchestrate nodes of a botnet, propagate an infection or collect stolen data in a remote command & control facility [14].

Unfortunately, each channel is characterized by a specific information hiding or steganographic injection mechanism, thus making the mitigation of such threats hard to abstract. Indeed, building a channel that encodes data by modulating the delay experienced between two packets requires a complete different detection scheme compared to techniques that directly inject information in the header of a specific protocol. Such a complexity is exacerbated by the availability of many protocols, each one characterized by a multitude of fields, optional values and functionalities often violating standard implementations, which eventually leads to a virtually unbounded amount of possibilities to conceal secret information [13, 15].

The usage of traditional network security appliances for detecting such kind of threats is therefore not straightforward. Specific requirements include both the methodology to detect anomalies and efficient technologies for getting visibility over network traffic in a very flexible and extensible way.

2.1 Covert channels in Network Packet Headers

Manipulating the value of protocol header fields is perhaps the simpler way to create covert channels. As a matter of fact, common protocols in both the network and transport layer (IP, TCP and UDP) have several fields that are seldom used, or have minor effects on protocol operation. As a paradigmatic example, an attacker could cloak information in unused header fields or develop an encoding scheme exploiting specific connection set-up/tear-down patterns, by altering the experienced packet loss or the distribution of bits in optional fields, as well as by creating sophisticated manipulation of ISO-OSI L7 protocols.

Table 1 lists the most popular covert channels targeting the TCP/IP stack grouped according to the protocol [15]. The range of potential vectors is rather broad, even if we limit our investigation to few protocols only. The size of the field and its impact on protocol operations are the main factors that make it appealing for attackers. Especially, the more bits are available the more information could be hidden. Similarly, changes to fields like the Flow Label in IPv6 and Reserved bits in TCP have very limited impact on common implementation of the TCP/IP stack. On the other hand, adding/replacing the Next Header or Payload Length in IPv6 is feasible, but requires significant manipulations of the packet structure.

Concerning the mitigation of network covert channels used to endanger the security of a network or to distribute malicious software, the literature does not offer a "one fits all" solution. The complexity

Table 1: Main known covert channels for TCP/IP.

Protocol	Field	Size (bits)
IPv4	TOS	8
	Time-To-Live	1
	Identification	Variable
	Fragment Offset	Variable
IPv6	Traffic Class	8
	Flow Label	20
	Hop Limit	1
	Next Header	–
	Payload Length	–
ICMP	Payload	–
TCP	Timestamps	1
	ACK number	32
	Reserved	3-4
UDP	Checksum	Variable

of network protocols as well as the heterogeneity of communication technologies prevent to develop a unified formal framework, as instead happens for channels targeting the single host [16]. According to a recent survey [17], detection of network covert channels is a "patchwork" of specific solutions leveraging a wide set of technologies, ranging from machine learning to support vector machines [18].

Partially ignited by the widespread diffusion of artificial intelligence, a recent trend for detecting and mitigating network covert channels is to look for anomalies in protocol operation. The idea is to inspect and collect information from protocol headers (e.g., fixed fields, optional values, status of a connection) in order to partially recover the lack of a *a priori* knowledge of the exploited carrier. Data can then be used to detect deviations from average values [8], to prepare suitable models (e.g., Markov templates to reveal the use of TCP connections to convey secret data [19]), to engineer "network pumps" able to process traffic and sanitize exploitable features [13], as well as to produce datasets for feeding AI-capable frameworks [17].

2.2 eBPF for Monitoring and Inspection

Several researchers have recently addressed the usage of eBPF for monitoring, inspection, and also enforcing at the network layer. Deri et al. [20] extend their existing tool *ntopng* with events generated by *libebpfflow*, a library that enriches network-layer data (e.g., source and destination IP addresses) with system metadata (e.g., source and destination processes and system users).

More often, eBPF programs are used to filter packets in the XDP, before the (costly) allocation of kernel data structures [21]. This is a typical approach for protecting against unwanted traffic, e.g., spoofed addresses or Denial-of-Service flooding attacks [22]. Independent studies have shown that XDP can perform four times better than common packet filtering tools [23]; this can be further improved by eBPF offloading to the hardware, using specific platforms as hXDP [24] and Netronome NICs [25]).

In the context of network tracing, Suo et al. [26] propose a framework where a master node translates user inputs into configuration files, eBPF agents are used to monitor network packets of specific connections at given tracepoints (e.g., virtual network interfaces), and measurements are centrally collected and analyzed. vNetTracer supports instrumenting kernel functions, return of kernel functions, kernel tracepoints and raw sockets through kprobe, kretprobe, tracepoints and network devices, hence providing a monitoring tool that works across the boundary of single domains.

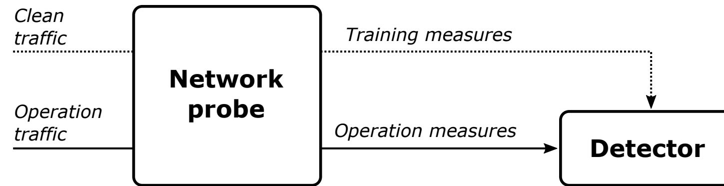


Figure 1: Reference framework for anomaly detection.

Cassagnes et al. [27] designed a system for deploying eBPF programs and collecting their measurements in containerized user-space applications. They used tools like Prometheus, Performance Co-Pilot, and Vector, and developed specific eBPF programs and the userland counterparts for monitoring the garbage collector, identifying HTTP traffic, and IP whitelisting. Nevertheless, they are only able to spot local information, and a global end-to-end view on distributed applications is still missing.

The first applications of eBPF programs for the detection of covert channels came only recently [8, 9]. It only covers IPv6 threats, but it clearly demonstrated the possibility to collect aggregate data suitable for the detection. Unfortunately, only preliminary performance indication was given, with no comparison with alternative technologies.

3 Detecting Covert Channels in Network Packet Headers

Given the wide range of potential covert channels (see Section 2), it is not surprising that the prevention, detection, and neutralization of threats exploiting network covert channels lead to a perpetual arm race between the attacker (which tries to exploit new carriers where to inject the secret data) and the defender (which seeks to inspect as many entities as possible to spot alterations).

Even if many researchers focused on the detection of specific patterns for creating covert channels, the most effective strategy to tackle the emergence of ever new techniques is the detection of anomalies from common data structures. In our work, we follow the common architectural pattern for anomaly detection, composed of a probe that takes measurements on the network traffic and a detector that spots anomalies by comparing measures taken offline as a reference (see Fig. 1).

The definition of the specific measurements is a key factor for both accuracy, performance, and scalability. Indeed, the plain knowledge of the values carried out by the vulnerable fields in the packet header leads to performance and scalability issues, which come from both the field length and the number of packets considered. Building on this consideration, we already introduced a new technique that scales well with the number of different communication flows and network packets, and also allows to tune the desired level of granularity [8].

Our approach is based on the estimation of the probability mass function for a given protocol field, by counting the number of occurrences of its different values. Indeed, to achieve better scalability with the field length, we split the whole space of possible values into B equally-spaced intervals, and keep a common counter for all values in the interval. By tuning the number of intervals, we can provide coarser- or finer-grained statistics to the detector.

Fig. 2 depicts our approach in a schematic way. We keep a set of B counters, one for each value interval. We usually denote these intervals as “bins,” to avoid confusion with the “time” interval to report measurements. A counter is incremented whenever the current value for the monitored field falls in the corresponding bin. A snapshot of these counters represents the measurement that we periodically report to the detector, i.e., a histogram of frequency for the values in each interval. After a reasonable amount of packets, this measurement gives a good approximation of the probability mass function. As an example, let us suppose to monitor the `Traffic Class` field of IPv6 (i.e., 2^8 possible values), by using

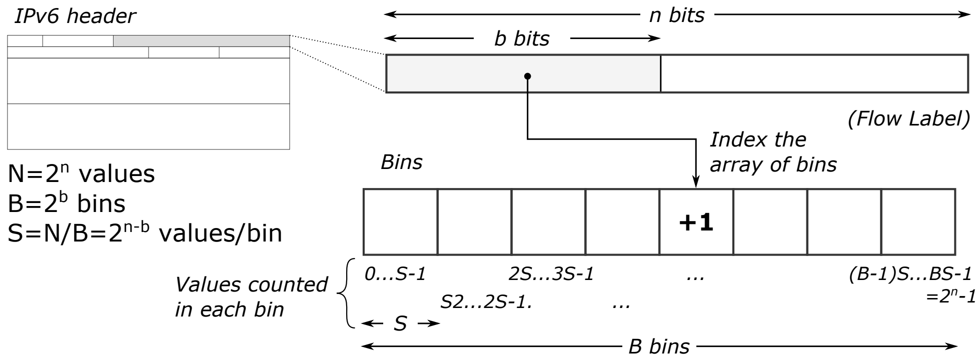


Figure 2: Mapping field values to bins.

only $B = 2^5 = 32$ bins. In this case, each bin corresponds to an interval of 2^3 values (i.e., $2^8/2^5$). When a packet with a `Traffic Class` set to 0 is seen, the counter of the first bin is incremented; instead, a packet bearing a `Traffic Class` of 18 will increase the counter of the third bin.

Since the identification of network covert channels requires to parse every network packet, we carefully considered scalability issues in the design of our solution. The main parameter in this respect is the number B of intervals, which directly determines the amount of memory to allocate for the counters. By using a smaller number of bins, we can reduce memory overhead; however, this might also smooth excessively the probability mass function, making the detection of anomalies impossible [9]. Memory allocation is instead independent of the number of packets, the number of flows, and (at least partially) the field size. The scalability in terms of CPU usage must be investigated experimentally, because it is difficult to be predicted analytically.

Apart for the mentioned benefits in terms of scalability, the proposed approach is flexible to be used for any kind of network traffic features, also beyond protocol header fields. Therefore, it is a good candidate to implement a generic framework for detecting a large number of different covert channels. In the following Sections we address in details performance implication to implement this kind of network probe with alternative technologies and tools.

4 More Efficiency for Monitoring and Inspection Processes

Traditionally, network appliances implement packet processing in hardware, because general-purpose architectures of desktop and server computers does not fit well the workflow for receiving, inspecting, and forwarding network packets. This approach has been largely used for cyber-security purposes as well, 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 ever new protocols and semantics, especially for cloud-native applications and service-oriented architectures [3].

A more flexible approach is required to effectively tackle the growing sophistication of attacks and ever-changing attack patterns. One typical problem is the need to continuously update the database of known signatures and rules, which have been the prevailing detection mechanisms for long. Besides, when the scope is narrowed to single applications or a few hosts, the amount of network traffic is not comparable with large infrastructures where hardware appliances are still needed. As a matter of fact, there are many software cyber-appliances, including open-source solutions, that today are largely used even in commercial and industrial environments, e.g., Suricata, Ossec, Snort, Zeek just to mention some of the most known tools available for free.

Detection of covert channels is not a standard feature of common tools available in the market, even if they usually provide some mechanisms to extend the basic capabilities and to define monitoring tasks tailored to different use cases.

Both Suricata and Zeek give access to a large number of protocol fields[2]. The Zeek scripting language is far more powerful than Suricata rules; however, Zeek scripts are interpreted at run-time and are not suitable for high packet rates. Moreover, according to its documentation, Zeek is not able to efficiently inspect fields that are present in each individual packet (e.g., IP headers).

Because of performance issues, cyber-appliances often leverage packet processing acceleration frameworks that *by-pass* the native kernel networking stack (which has been designed to support the larger number of even outdated protocols), and give direct access to hardware queues and functionalities in Network Interface Cards (NICs), e.g., PF_RING, Netmap, DPDK, OpenOnLoad. For instance both Zeek, Suricata and nProbe can use plain PF_RING and its extensions, but their effectiveness in software environments is questionable [4]. First, the main processing code is usually developed in user-space, because this is simpler for programmers and 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, 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 a few table look-ups. However, deep packet inspection requires a lot of parsing on the packet data, not to mention the need for continuous polling the NIC, which leads to large wasting of CPU time [4].

Finally, we argue that the typical size of cyber-security appliances does not fit the agile and lightweight nature of modern cloud-native applications. These tools 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 today [1]. As a concrete example, we can think of Kubernetes. In this case, complex applications are decomposed in many elementary services, each one run into a dedicated container. There is no a-priori knowledge of which node in the cluster will host each container (unless specific constraints are given, but this is not the common option for several management reasons), and the topology of the application may change at run-time due to management actions (scaling, migration, resiliency). Deploying full-featured cyber-security appliances in each pod for monitoring the main business logic container brings a large overhead: for instance, Zeek would add 307 MB to a base image of 124 MB (official Debian testing), yielding an overhead of 247%! On the other hand, running a Zeek instance for all pods hosted on the same node may be unacceptable in case of multi-tenancy (i.e., public cloud), not to mention the additional overhead to bind packets to their pod and application.

5 Monitoring Technologies

Implementing monitoring processes to detect covert channels is therefore not trivial, especially when modern computing paradigms are taken into consideration (cloud-native applications, service-oriented architectures, IoT). We address this issue by comparing alternative tools that could be used for this purpose. Our work takes into consideration the following technologies:

- an existing general-purpose tool, **Zeek**, selected because of its flexibility and extensibility by a simple scripting language;

[2]See, for instance, the list of available keywords for Suricata rules: `https://suricata.readthedocs.io/en/latest/rules/index.html`, and the list of protocol analyzers for Zeek: `https://docs.zeek.org/en/master/script-reference/proto-analyzers.html#zeek-dns`.

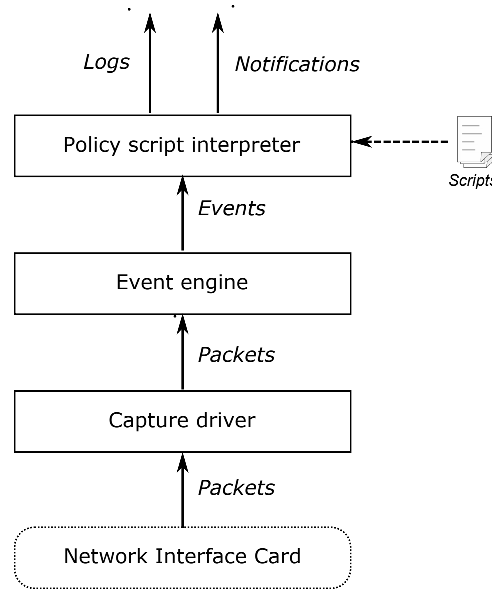


Figure 3: Zeek architecture and processing model.

- a raw implementation in the C language based on **libpcap**, which is the common utility used by many tools to get packets from the kernel;
- a more innovative approach that leverages the **eBPF** framework, which lays the foundation for the automatic generation of inspection code at run-time.

The main purpose of our investigation 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.

5.1 Zeek-stego

Zeek (formerly Bro) is a fully open-source tool optimized for interpreting network traffic and generating logs based on that traffic. It enables collection of at least two, and in some ways three, of these data forms, namely transaction data, extracted content, and alert data. However, it is not designed as a complete Intrusion Detection System (IDS), hence it is not optimized for byte matching; Suricata and Snort better fit this need. In addition, it is not either a protocol analyzer, like tcpdump, seeking to depict every element of network traffic at the frame level, or a system for storing traffic in packet capture (PCAP) form. Zeek is typically used to ship logs to Security and Information Event Management (SIEM) platforms, hence playing the role of network probe in our reference architecture.

The Zeek architecture includes a capture driver (libpcap, raw socket, PF_RING are the most common options), which delivers packets to the Event engine (see Fig. 3). This is the core of the Zeek framework, which 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, HTTP request/response, etc. Semantics related to the events are derived by Zeek's second main component, the script interpreter, which executes a set of event handlers written in Zeek's custom scripting language and generates logs and notifications for some external consumer (user or SIEM).

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, Turing-complete scripting language for

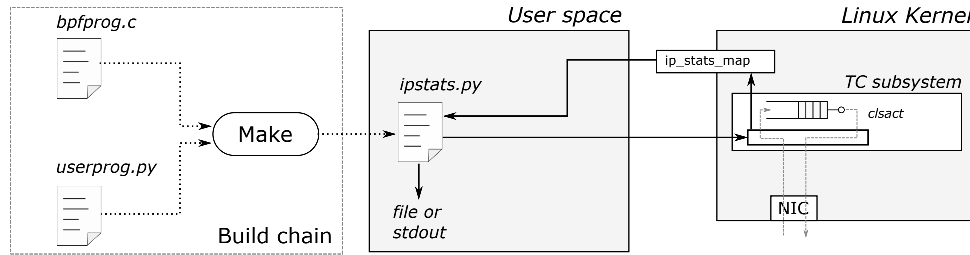


Figure 4: Architecture of the bccstego framework.

expressing arbitrary analysis tasks. Indeed, even Zeek's default analyses, including logging, are done via scripts; no specific analysis is hard-coded into the core of the system.

Following the above model, it is rather simple to create custom scripts for the generation of the measurements described in Sec. 3. 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 that is able to report relevant fields of the IPv4/v6 headers we are interested in. Once the event is available, we use the scripting language to catch it and create the same histogram of eBPF programs. The source code of the patch, scripts, and even a docker file are available in github[3].

5.2 Libpcap

The second approach we decided to use leverages libpcap, a C/C++ library developed for Unix-like operating systems as a part of the TCPDump[4] packet analyzer which enables traffic monitoring. The library allows to retrieve and capture packets from a live network device in a simple and straightforward manner.

Our implementation 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 (e.g., B or the considered field) and updating the counters of the data structure described in Sec. 3.

The current version considers almost all the fields contained in Table 1 except for the timestamps of TCP packets and the ICMP protocol. The implementation can be easily extended to consider other protocols and fields by simply adding cascading if-conditions in the main function.

5.3 Bccstego

Bccstego is the preliminary kernel of a framework that creates eBPF programs at run-time for monitoring and inspection purposes. The main use case is currently the detection of network covert channels. The underlying concept is to share a common pattern for parsing packets and collecting data, while different eBPF programs are developed for specific protocols or steganographic threats.

Fig. 5.3 shows the current architecture of the framework. Starting from a common eBPF template (`bpfprog.c`) and a skeleton of the corresponding userspace control application (`userprog.py`) we generate a single Python executable (`ipstats.py`). This approach simplifies the deployment, installation, and usage on multiple systems.

The `bpfprog.c` template contains all necessary instructions for parsing IPv4/6, TCP and UDP protocol headers, and includes some placeholders to read specific fields and update the frequency histogram described in Sec. 3. The userland application replaces these placeholders with appropriate code snippets, according to the protocol field requested by the user. This way, the same program can be easily

[3]`https://github.com/mattereppe/zeek-stego`.

[4]`https://www.tcpdump.org`

extended to cover additional fields, as the need arises; in addition, the number of instructions in the eBPF program is minimized, and this is important for performance reasons, because the eBPF code is run for each packet.

Our implementation of the user-space application is based on the BCC framework[5], which provides a Python class for compiling the code, loading it into the kernel, and retrieving data. This facilitates the overall management of eBPF programs, although the usage of Python increases resource usage.

The current implementation covers all vulnerable fields listed in Table 1 for IPv4/6, TCP, and UDP, hence notably extending the original version which supported IPv6 only [10]. So far, we only miss ICMP inspection. The code is available in github[6].

6 Complexity analysis

Before considering performance, it is worth elaborating on the difficulty to extend the tools under consideration. This is an important aspect for covert channels, because a holistic solution is almost impossible to be implemented, for the many reasons outlined in Sec. 2.

Zeek is a well-known tool, and its scripting language is simpler to learn than getting acquainted with eBPF programming. The documentation is available online, with some basic examples; further, the system comes with a large set of pre-built functionality (the "standard library"), which can be used as more concrete examples by the beginners. The main limitation here is that the scripting language can only process events generated by the core platform, which is not as much well documented. This flaw has been partially addressed by the Zeek developers. Recent Zeek versions feature Packet Analysis: one could (re)write his own packet analyzer for IP and include the necessary code for generating an event for each IP packet, which reports the content of the monitored fields. We leveraged this feature for our extension, even if it is mostly conceived to parse new protocol headers rather than extending existing ones. There is some documentation available,[7] but the process is far more complex than working with scripts. Finally, since events are eventually processed by a script interpreter, processing is likely to become overwhelming.

The development of plain C code is the easier approach for a skilled developer; capturing packets with libpcap is rather simple and straightforward, well documented, and full of working examples. Of course, the developer has to create the necessary code for managing threads, polling, and whatever else is needed to handle the asynchronous reception and processing of network packets. Even if basic C skills are quite simple to learn, more advanced programming constructs may be discouraging for a beginner. More important, in this case extensions 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.

Finally, writing eBPF programs is not easy. A good knowledge of C is required, but there are some challenging issues:

- the relationship and interaction between userland utilities and eBPF programs must be fully understood, and this is not easy, given the poor and largely fragmented documentation available;
- eBPF programs have a small stack size available, which limits the number of functions and instructions; this practically limits the protocols that can be parsed with a single program;

[5]BPF Compiler Collection, available on line: `https://github.com/iovisor/bcc`. Last Accessed: July 2021.
[6]`https://github.com/mattereppe/bccstego`.
[7]`https://docs.zeek.org/en/master/frameworks/packet-analysis.html`.

- eBPF programs are checked by a kernel verifier before being loaded. It ensures there are no loops and the program always terminates, but, on the other hand, it still has many limitations, including high rate of false positives, poor scalability, and lack of support for loops [28]. As a matter of fact, the verifier does not scale to programs with a large number of paths, its algorithm is not formally specified, and no formal argument about its correctness is given.

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

7 Performance evaluation

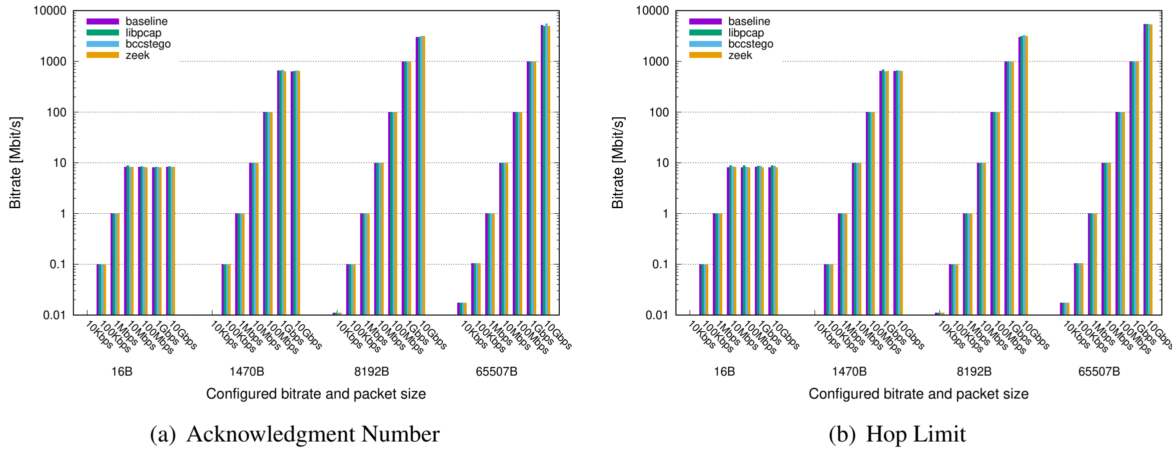
To conduct experiments, we prepared a testbed composed of three virtual machines (VMs), hosted on an OpenStack installation. All nodes ran on the same hypervisor, 2x Intel Xeon CPU E5-2660 v4@2.00GHz with 14 cores and hyperthreading enabled, 128 GB RAM, 64 GB SSD storage. Two VMs were used as traffic source and destination, whereas the third was used as router in between and hosted the monitoring tool (i.e., Zeek, libpcap filter and bccstego 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 VMs ran Debian GNU/Linux 11 with kernel 5.10.

Performance evaluation was intended to understand the impact of the alternative implementations on network operation, since the effectiveness of the proposed monitoring mechanisms was already demonstrated in a previous paper [8]. To this purpose, we considered both UDP and TCP streams generated with `iPerf3`[8] while changing the following parameters:

- Packet size for UDP: 16, 1470, 8192, and 65507 bytes, to account for the minimal packet size, and Maximum Transfer Unit (MTU) 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;
- Maximum Segment Size (MSS) for TCP: 88, 536, 1460 and 9216 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`[9] (System Activity Report) utility for CPU (e.g., user-space, kernel-space, etc.) and `pmap`[10] 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 the TCP and IP header. 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

[8] `https://iperf.fr`
[9] `https://linux.die.net/man/1/sar`
[10] `https://linux.die.net/man/1/pmap`



(a) Acknowledgment Number (b) Hop Limit

Figure 5: Measured bitrate at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

took into account two different amount of bins, i.e., 2^{12} and 2^{8}, to understand their impact in the overall performances.

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.

7.1 Impact on packet transmission

We considered how monitoring operations affect the transmission of network packets, by measuring the transmission rate, packet error rate and jitter. These figures were taken from logs generated by `iPerf3`.

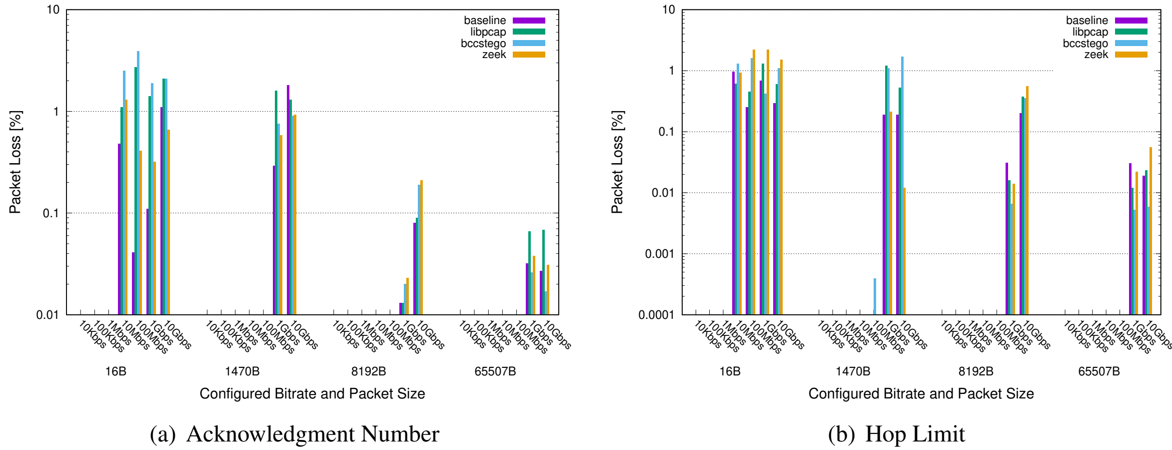
Fig. 5 shows how the bitrate for UDP flows changes at the receiver while varying both the packet size and the transmission bandwidth, in case of inspection of the `Acknowledgment Number` and the `Hop Limit`. As expected, smaller packet size leads to lower bitrate, but there are not meaningful differences with respect to the baseline when using the proposed monitoring mechanism, neither for inspection of the `Acknowledgment Number` (Fig. 5(a)) nor of the `Hop Limit` (5(b)). We can also notice that in our setup it is impossible to transmit at 10 Gbit/s, even with the largest packet size.

Fig. 6 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 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.

Fig. 7 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.

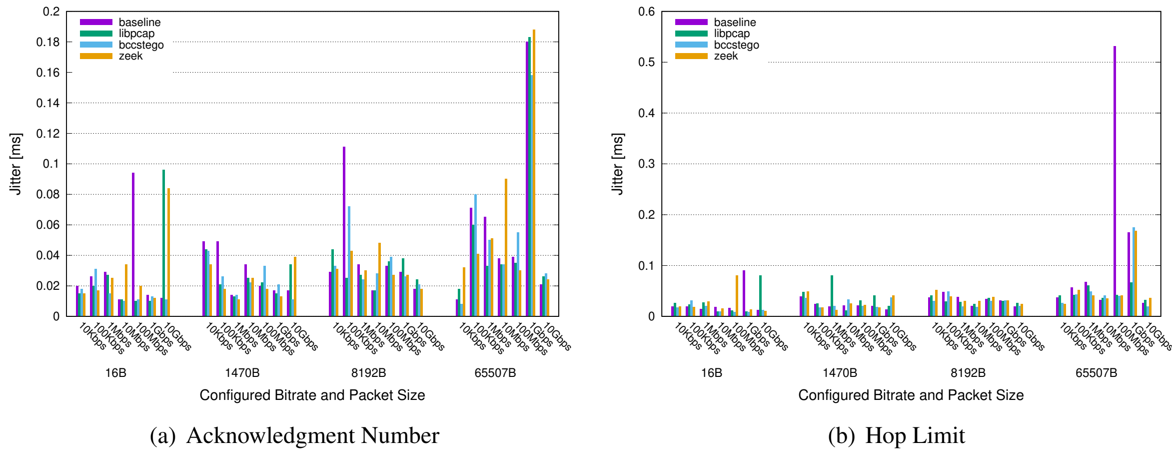
For TCP flows, we measured the average transmission bitrate while varying the MSS (see Fig. 8). In general, a larger MSS results in higher bitrate, because the TCP congestion control protocol works better. No significant differences can be seen for inspection of the different fields or the different monitoring mechanisms.

In conclusion, the measurements required by our detection techniques do not impact 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.



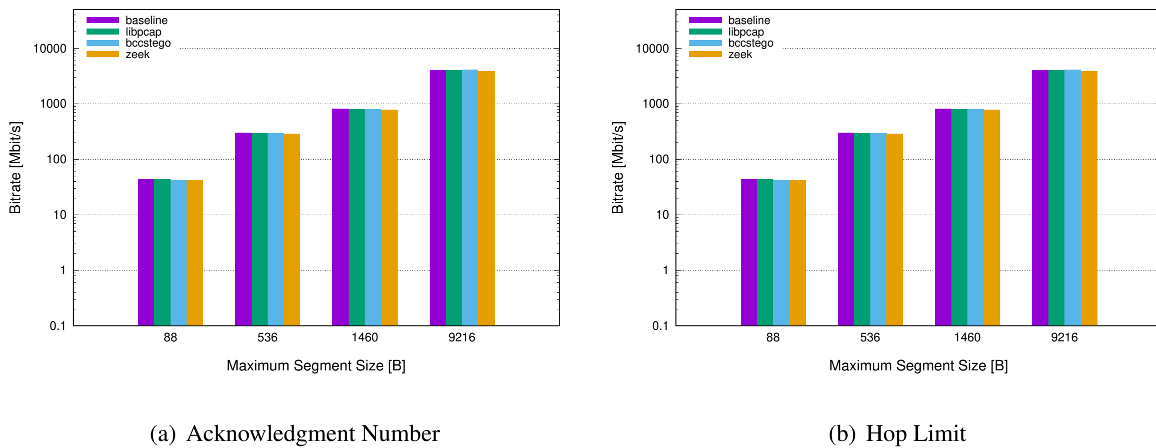
(a) Acknowledgment Number (b) Hop Limit

Figure 6: Measured packet loss at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



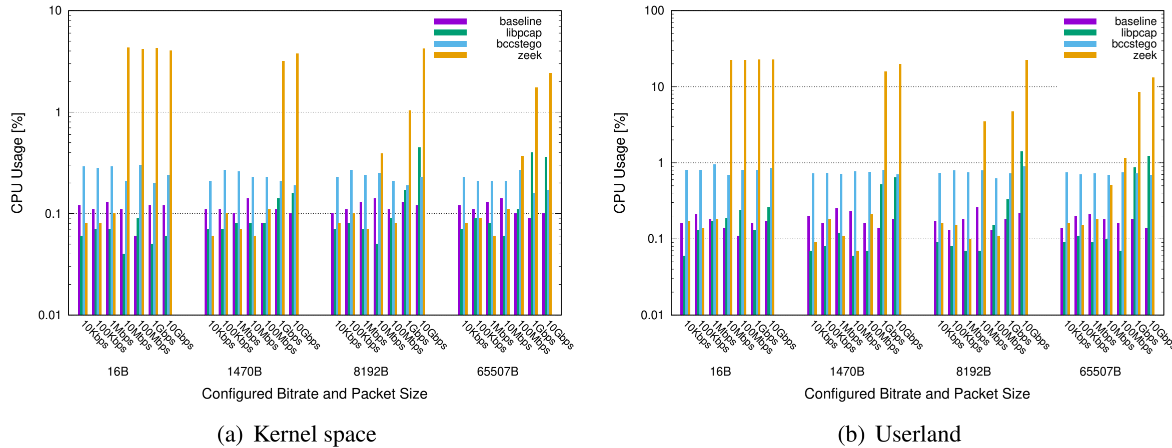
(a) Acknowledgment Number (b) Hop Limit

Figure 7: Measured packet jitter at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



(a) Acknowledgment Number (b) Hop Limit

Figure 8: Measured bitrate at the receiver, while varying the MSS for a TCP flow.



(a) Kernel space (b) Userland

Figure 9: 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.

7.2 CPU usage

We measured the CPU usage for the different technologies. Since Zeek and our 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 bccstego which leverages in-kernel eBPF programs. Figs. 9 and 10 show the CPU consumption in case the `Acknowledgment Number` or the `Hop Limit` are monitored, respectively.

As expected, both Figs. 9(a) and 10(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 bccstego 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).

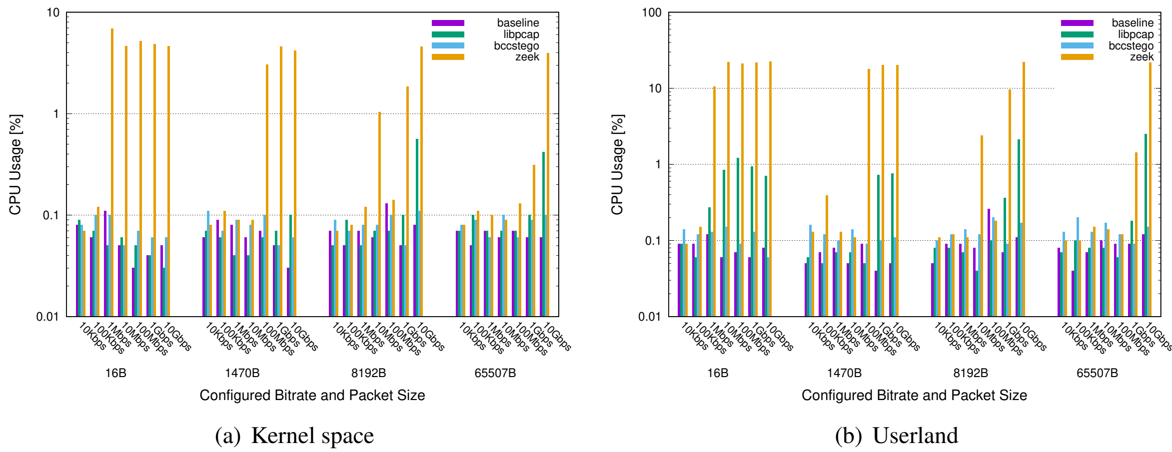
Zeek also requires more CPU in the user-space (i.e., 20% in the worst case), with respect to the other tools. Instead, bccstego performs quite well, remaining close to the baseline in almost all scenarios.

Fig. 11 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 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 Acknowledgment Number in this case, since data for the Hop Limit leads to similar consideration and is anyway reported in Fig. 10.

These considerations can also be extended for the case of TCP flows while varying the MSS. Figs. 12 and 13 show the kernel-space, the user-space and the cumulative CPU used by all the tools.

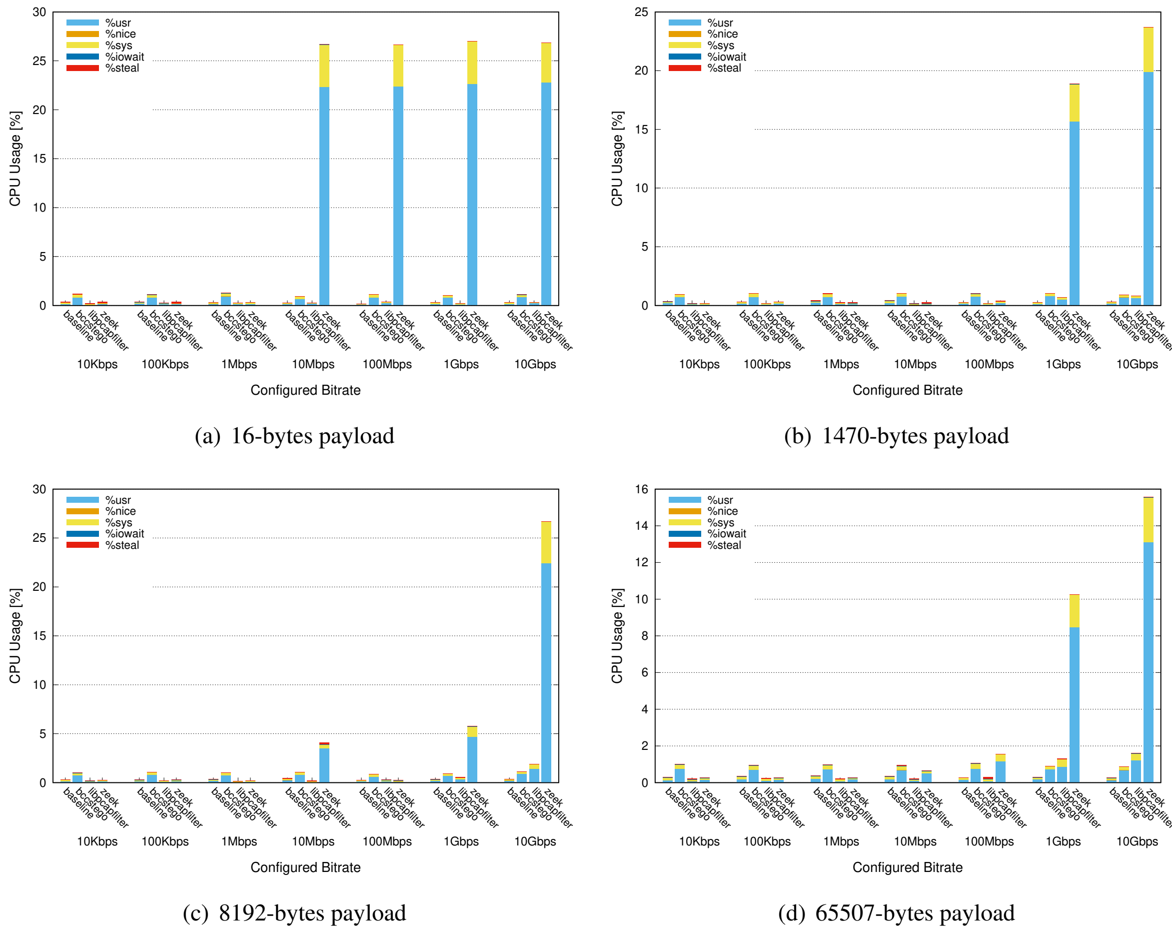
7.3 Memory allocation

Finally, Fig. 15 shows the amount of memory allocated by each technology. We considered a breakdown of memory allocation for the Virtual Memory Size (VMS), Resident Set Size (RSS), Proportional Set Size (PSS) and Anonymous (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



(a) Kernel space (b) Userland

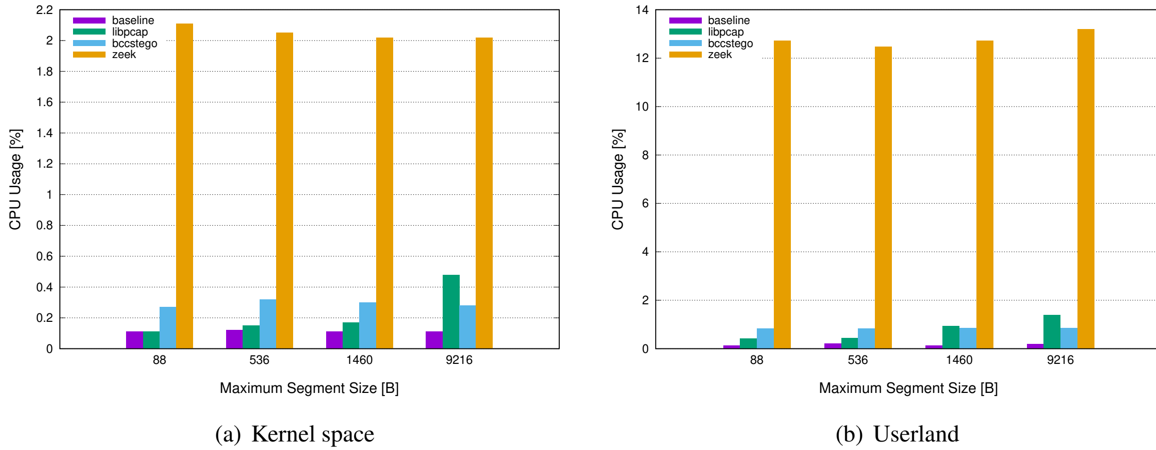
Figure 10: CPU usage measured at the intermediate node, while varying the packet size and transmission bitrate for a UDP flow. The monitored field is the Hop Limit.



(a) 16-bytes payload (b) 1470-bytes payload

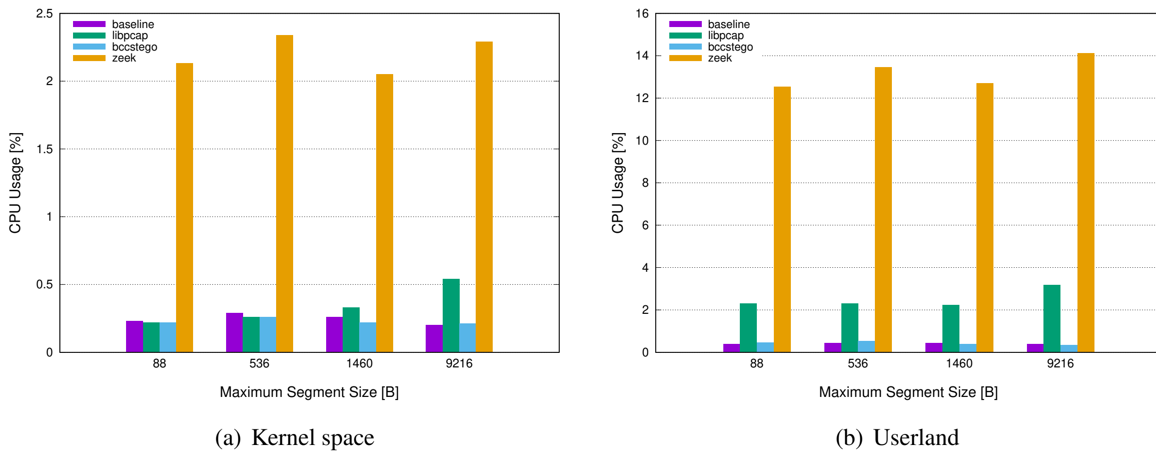
(c) 8192-bytes payload (d) 65507-bytes payload

Figure 11: Cumulative CPU usage measured at the intermediate node, for a UDP flow. The monitored field is the Acknowledgment Number.



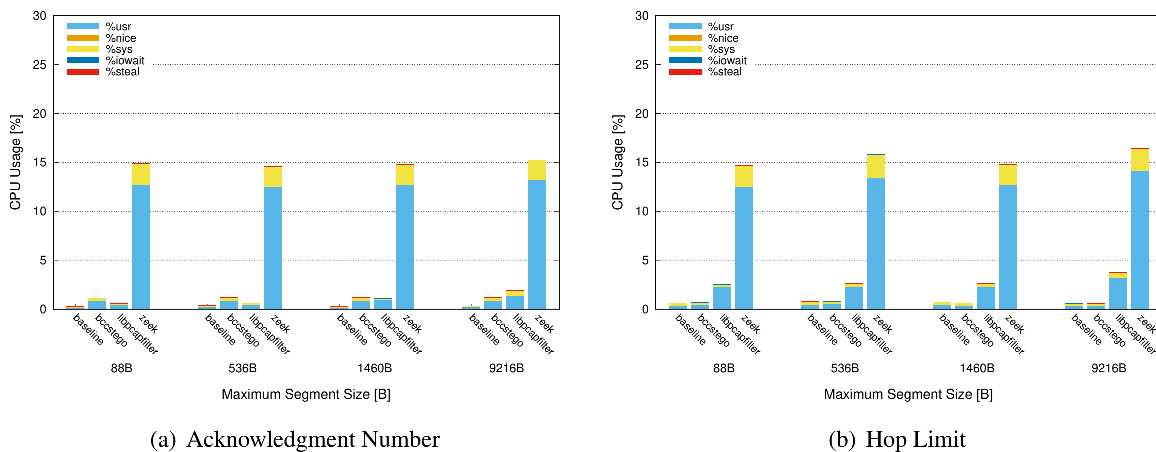
(a) Kernel space (b) Userland

Figure 12: CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is the Acknowledgment Number.



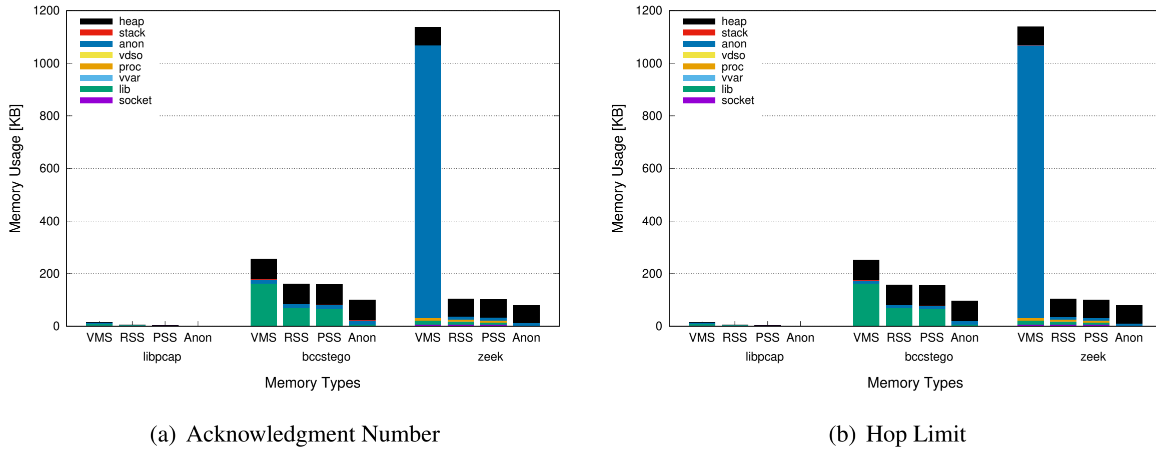
(a) Kernel space (b) Userland

Figure 13: CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is the Hop Limit.



(a) Acknowledgment Number (b) Hop Limit

Figure 14: Cumulative CPU usage measured at the intermediate node, for a TCP flow.



(a) Acknowledgment Number

(b) Hop Limit

Figure 15: Memory allocation for the different inspection tools.

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 Acknowledgment 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.

8 Conclusion and future work

In this paper we analyzed different implementations for a lightweight packet inspection mechanism that aims at detecting network cover channels. Specifically, we showcased how it is possible to collect network information in an efficient counter-based data structure.

Among the technologies used to implement this mechanism, the results for the eBPF-approach indicate a negligible impact on packet transmission, and this is a very good result, since flow monitoring applications usually do not scale well with the transmission rate. Of course, additional evaluation is necessary in case of physical systems and higher bitrates, which currently do not fall under the scope of our work.

In terms of resource usage, eBPF programs demonstrate to be a very powerful, extensible and scalable solution for packet inspection. 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).

Our future work will address the need for automatic code generation, in order to provide more flexibility in the type and number of fields to be monitored. We also plan to improve the detection mechanism beyond plain rule-based comparison with a given threshold, for instance by leveraging machine learning or another form of artificial intelligence.

Acknowledgments

This work was supported in part by the European Commission under Grant Agreements no. 786922 (ASTRID), no. 833456 (GUARD), and no. 833042 (SIMARGL).

References

[1] M. Repetto, A. Carrega, and R. Rapuzzi. An architecture to manage security operations for digital service chains. *Future Generation Computer Systems*, 115:251–266, February 2021.

[2] D. Soldani and A. Manzalini. On the 5g operating system for a true digital society. *IEEE Vehicular Technology Magazine*, 10(1):32–42, March 2015.

[3] R. Rapuzzi and M. 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.

[4] M. Repetto and A. Carrega. Efficient flow monitoring for virtualized applications with ebpf. Technical report, ASTRID project, July 2021. `http://doi.org/10.5281/zenodo.5113889.` [Online; accessed on December 21, 2021].

[5] S. Miano, F. Risso, M. Vásquez Bernal, M. Bertrone, and Y. Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transaction on Network and Service Management*, 18(1):133–151, March 2021.

[6] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, and S. Zander. The new threats of information hiding: The road ahead. *IT professional*, 20(3):31–39, May/June 2018.

[7] D. Llamas, C. Allison, and A. Miller. Covert channels in internet protocols: A survey. In *Proc. of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting (PGNET'05), Liverpool, UK*. Liverpool John Moores University School of Computing & Mathematical Sciences, June 2005.

[8] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli. Kernel-level tracing for detecting stegomalware and covert channels in linux environments. *Computer Networks*, 191, May 2021.

[9] L. Caviglione, M. Zuppelli, W. Mazurczyk, A. Shaffhauser, and M. Repetto. Code augmentation for detecting covert channels targeting the ipv6 flow label. In *Proc. of the 2021 IEEE International Conference on Network Softwarization (NetSoft'21), Tokyo, Japan (Virtual)*, pages 450–456. IEEE, June 2021.

[10] M. Repetto, L. Caviglione, and M. Zuppelli. bccstego: A framework for investigating network covert channels. In *Proc. of the 16th International Conference on Availability, Reliability and Security (ARES'21), Vienna, Austria*, pages 1–7. ACM, August 2021.

[11] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[12] W. Mazurczyk and L. Caviglione. Cyber reconnaissance techniques. *Communications of the ACM*, 64(3):86–95, March 2021.

[13] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, September 2007.

[14] W. Mazurczyk and L. Caviglione. Information hiding as a challenge for malware detection. *IEEE Security & Privacy*, 13(2):89–93, April 2015.

[15] A. Mileva and B. Panajotov. Covert channels in tcp/ip protocol stack-extended version. *Central European Journal of Computer Science*, 4(2):45–66, June 2014.

[16] R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[17] L. Caviglione. Trends and challenges in network covert channels countermeasures. *Applied Sciences*, 11(4):1641, February 2021.

[18] T. Sohn, J. Seo, and J. Moon. A study on the covert channel detection of tcp/ip header using support vector machine. In *Proc. of the 2003 International Conference on Information and Communications Security*

(ICICS'03), Huhehaote, China, pages 313–324. Springer, October 2003.

[19] J. Zhai, G. Liu, and Y. 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'10), Nanjing, China*, pages 893–897. IEEE, November 2010.

[20] L. Deri, S. Sabella, and S. Mainardi. Combining system visibility and security using ebpf. In *Proc. of the 3rd Italian Conference on Cyber Security (ITASEC'19), Pisa, Italy*, volume 2315, pages 50—-62. CEUR-WS, February 2019.

[21] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. 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'18), Heraklion, Greece*, page 54–66. ACM, December 2018.

[22] G. Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Proc. of the 2017 Technical Conference on Linux Networking (Netdev 2.1), Montreal, Canada*, pages 226–231, April 2017.

[23] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. Performance implications of packet filtering with linux ebpf. In *Proc. of the 30th International Teletraffic Congress (ITC'18), Vienna, Austria*, pages 209–217. IEEE, September 2018.

[24] M. Spaziani Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hxdp: Efficient software packet processing on fpga nics. In *Proc. of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20), Online*, pages 973–990. USENIX Association, November 2020.

[25] Netronome. Avoid kernel-bypass in your network infrastructure, January 2017. `https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/` [Online; accessed on December 20, 2021].

[26] K. Suo, Y. Zhao, W. Chen, and J. Rao. vnettracer: Efficient and programmable packet tracing in virtualized networks. In *Proc. of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS'18), Vienna, Austria*, pages 165–175. IEEE, July 2018.

[27] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State. The rise of ebpf for non-intrusive performance monitoring. In *Proc. of the 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS'20), Budapest, Hungary*, pages 1–7. IEEE, April 2020.

[28] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. 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'19), Phoenix, Arizona, USA*, pages 1069–1084. ACM, June 2019.

Author Biography

Marco Zuppelli is a second year PhD student at University of Genoa and a research fellow at the Institute for Applied Mathematics and Information Technologies of the National Research Council of Italy. Within the European Project SIMARGL (Secure Intelligent Methods for Advanced RecoGnition of malware and stegomalware), he investigates novel detection methods for steganographic malware exploiting both network and local covert channels. His main research interests are the use of in-kernel methodologies (e.g., the extended Berkeley Packet Filter) to collect information on software/network components, and the design of mechanisms for detecting malicious communications in an efficient, scalable and extensible manner.

Alessandro Carrega is a Software and Network Engineer. He is currently a Senior Researcher at the National Inter-University Consortium for Telecommunications (CNIT) located in the Genoa (Italy) research unit. He took part in the activities of many national and European projects (e.g., H2020 ASTRID, GUARD, ARCADIA and INPUT, FP7 IP ECONET, PRIN EFFICIENT, FIRB GreenNet, and FIWARE) and he is an active reviewer for many different international journals and conferences (IEEE and ACM). He has co-authored several papers in international conference proceedings. In 2010, he won the best paper award at the 3rd Int. Workshop on Green Communications (GreenCom 2010) co-located with the IEEE GLOBECOM Conference. In 2011 he was a Visiting Ph.D. Scholar at Portland State University (PSU), Portland, OR, USA under the supervisor of Prof. Suresh Singh as an active member of the FINE2 Italia – USA collaboration project. He received the B.S. and M.S. degrees in Computer Engineering and the Ph.D. degree in Green Networking from the University of Genoa in 2005, 2007 and 2013, respectively. Alex Carrega's research is on networking (energy-aware, performance optimization, and virtualization), software routers, NFV and SDN (OpenFlow), container-orchestration system for automating computer application deployment, scaling, and management (Kubernetes) and cloud computing platforms (Openstack). Finally, he is involved in collaboration with many industries, such as Telecom Italia, Broadcom, Nokia, Ericsson, Huawei, etc., and industrial fora, like GeSI.

Matteo Repetto received the Ph.D. degree in Electronics and Computer Science in 2004 from the University of Genoa. From 2004 to 2009 he was a postdoc at University of Genoa. From 2010 to 2019 he was a Research Associate at CNIT. In 2019 he joined the Institute for Applied Mathematics and Information Technologies (IMATI), CNR. He has been teaching many courses in telecommunication networks and network security. He has been involved in several research national and international projects on quality of service, mobility in data networks, energy efficiency, cloud computing, and network function virtualization. He has been the scientific and technical coordinator of the ASTRID (www.astrid-project.eu) and GUARD (https://guard-project.eu/) projects, which investigate new security paradigms for cloud services. He has co- authored over 60 scientific publications in international journals and conference proceedings. His current research interests include pervasive communications and mobility management, energy-efficient networking, software-defined networking and network function virtualization, cloud/fog/edge computing and network security.