Efficient, portable and extensible packet inspection with eBPF

Matteo Repetto, Marco Zuppelli, and Alessandro Carrega

***Abstract*—The availability of virtualization technologies and cloud models has made possible an effective decoupling of applications and services from the underlying infrastructure, which eventually allows far more flexibility in deployment and operation processes than in the past. However, this also means that hardware acceleration will be available ever more seldom, which may jeopardize the efficiency of computing-intensive tasks, including network monitoring and packet inspection in cyber-security appliances.**

In ASTRID, we investigated the usage of the extended Berkeley Packet Filter (eBPF) for effective and efficient packet inspection. Our goal is the implementation of a tool that provides similar information as existing cyber-security appliances but with a reduced execution footprint, in order to be easily integrated in cloud-native applications without any hardware or software dependency on the underlying infrastructure. In this paper, we discuss the main results of our work with respect to two challenging use cases, namely amplification attacks and network covert channels.

***Index Terms*—Network covert channels, cloud computing, eBPF, amplification attacks, Kubernetes**

I. Introduction

New computing models are today available that leverage virtualization, ubiquitous connectivity and multi-tenancy. Originally conceived to improve the management and efficiency of data centers, the cloud paradigm has been progressively extended to the edge of access networks and to personal devices, building a pervasive computing continuum that brings unprecedented flexibility in placing and deploying distributed applications. Beyond cost reduction by replacing physical appliances with (cheaper) commodity hardware and more agile management processes with software-defined infrastructures, this new opportunity has also accelerated the transition from monolithic and rigid software architectures to micro-services and service meshes, which helps place the different software components into multiple infrastructures and locations, according to cost, proximity, capacity, and many other types of constraints. This evolution has not only interested typical data-center applications, but has also extended to different domains, being Network Function Virtualization the most notable example.

Despite of the well-known benefits brought by cloud paradigms, the main drawback is represented by the increasing difficulty in managing security processes [1]. Even though most of existing cyber-security appliances may be realized entirely in software and easily re-used with the Infrastructure-as-a-Service (IaaS) model, this is not straightforward when the deployment involves multiple infrastructures and when different cloud models are used [2]. As a matter of fact, all major cybersecurity vendors have already included specific tools for the cloud in their products portfolio since many years, but they often rely on specific agents and capabilities of the virtualization infrastructure. In this respect, a transition from infrastructure-centric to more service-centric architectures for monitoring and inspection is desirable, which give better visibility over the workload itself and be agnostic of the underlying execution environment [3].

In ASTRID we tackled this challenge for the network traffic, which is among the main attack vectors. The most common network threats are still (Distributed) Denial of Service (DDoS), also leveraging amplification from buggy services in the Internet [4], and attacks against web services and public APIs[1]; more recent threats come from network covert channels, which are stealthy communications to hide the presence of malware in compromised systems [5]. Our main objective was to make the inspection process lightweight, not bound to specific hardware/infrastructure capabilities, and easily extensible to cope with the ever-evolving attack patterns and threat landscape.

Our work leverages the extended Berkeley Packet Filter (eBPF), a Linux native technology for instrumenting the kernel at run-time. Even if it has been used for many years for pinpointing performance issues, its application for security purposes is also gaining interest in the recent years, as witnessed by the increasing number of tools that make use of this technology (e.g., Suricata, Sysdig Falco, Cilium). Since there is not a standard management framework for eBPF, in ASTRID we adopted *Polycube*[2], which can be used to create network service chains in the NFV domain [6]. Nevertheless, as part of our exploitation plan we also developed a standalone tool, named *bccstego*[3], which is more portable than Polycube to other use cases than NFV.

We compare our approach with a well-known and largely used network monitoring tool, namely *Zeek*[4], which can be used to implement the same kind of processes our use cases need. Zeek (formerly Bro) is fully open-source and provides a flexible framework that is easy to extend; in this case, we replicated the same features provided by our eBPF programs

M. Repetto and M. Zuppelli are with CNR – Institute for Applied Mathematics and Information Technologies (IMATI).

A. Carrega is with CNIT – S2N National Lab.

[1]Application Programming Interfaces are commonly exposed today to provide remote services; they are often implemented over HTTP(s).

[2]https://polycube.network/.

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

[4]https://zeek.org/.

in the Zeek language. For a subset of the considered use cases, we also include a plain implementation in C code.

The rest of the paper is organized as follows. Section II highlights the main innovations of our work. Section III briefly introduces Polycube and the dynmon service developed in ASTRID, whereas a brief introduction to bccstego is given in Section IV. We describe the eBPF programs for our specific use case in Section V, and then conduct deep performance evaluation in Section VI. Finally, we give our conclusion in Section VIII.

II. ASTRID INNOVATIONS

One of the main objective for the ASTRID project was to follow an infrastructure-agnostic approach, extending the scope of cloud workload protection platforms. We developed a set of eBPF programs addressing packet inspection needs of our network-oriented use cases. In particular, we demonstrated the following **main innovations**:

- efficiency: deep packet inspection is possible (at least for network headers) with a smaller execution footprint than existing applications;
- extensibility: kernel can be instrumented at run-time from the user-space in a safer way than with modules;
- portability: packet inspection without hardware or software acceleration at the maximum speed allowed by a virtualized infrastructure.

When combined together, these innovations allow service operators more freedom in the selection of the environment(s) to run their applications, both in terms of cloud models and virtualization technologies. This is especially important for cloud-native applications, which run in containers and cannot add custom features to the running kernel.

In the rest of this Section, we elaborate on each innovation separately.

A. Efficiency

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.

With the growing adoption of softwarization and virtualization techniques, hardware appliances can be seldom used, because cloud applications are often designed to work in different environments. Besides, the amount of network traffic processed by a typical virtual service is not comparable with physical infrastructures. In case of network services, the solution is either to load balance the traffic among multiple replicas of network functions, or to make usage of hardware acceleration functions offered by the underlying infrastructure. The last option is perfectly acceptable for packet forwarding, which only considers a few fields of the lower protocol headers; however, it is more challenging for security mechanisms, which usually require deeper analysis and look for ever-changing patterns.

To overcome the well-known limitations of existing built-in networking stacks of common operating systems, which have been largely designed to support the larger number of (even outdated) protocols, a number of technologies and frameworks have been proposed to *by-pass* this native implementation, and to give direct access to hardware queues and functionalities in Network Interface Cards (NICs), e.g., PF_RING, Netmap, DPDK, OpenOnLoad. Kernel by-pass functions are implemented in the kernel, but 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. Kernel bypass technologies have been already integrated in cyber-security appliances; for instance both Zeek, Suricata and nProbe can use plain PF_RING and its extensions, but their effectiveness in virtualized environments is questionable [7].

Although kernel by-pass is able to process packets at line rate for up to 10 Gbps Ethernet links[5], there are several issues with cloud and NFV deployments [8]. As a matter of fact, the aforementioned technologies usually require the exclusive allocation of resources (i.e., CPU cores) to achieve good performance; this is perfectly fine when we have machines dedicated to networking purposes but it becomes overwhelming when this cost has to be paid for every server in the cluster since they permanently steal precious CPU cycles to other application tasks.

Our investigation shows that, in a virtualized environment, the usage of eBPF for packet inspection performs better than existing tools, both in terms of speed and CPU/memory usage. This is not surprising, because eBPF programs are executed natively on the target platform[6], which provides a notable speed-up compared to an interpreted execution. Differently from other frameworks (e.g., DPDK), which mitigate the overhead of hardware interrupts by continuously polling the NIC for new packets, eBPF programs are only triggered when a packet is received. Therefore, a few CPU cycles are added to the Kernel built-in processes, while avoiding to consume almost all CPU in the absence of traffic.

Our analysis shows that our inspection tasks put an almost negligible overhead on packet processing; of course, this is also due to the ASTRID concept of moving as much aggregation and detection logic outside single agents, hence we can say that the original objective has been largely achieved.

ASTRID agents can be replicated in all necessary virtual functions without incurring in large overhead, which is useful when there is the need for intra-pod monitoring or cross-cloud deployments. This is important to reduce the usage of CPU, memory and disk space, in order to make the deployment process faster and to decrease the cost in case public cloud infrastructures are used.

[5]https://www.ntop.org/nprobe/10-gbit-line-rate-netflow-traffic-analysis-using-nprobe-and-

[6]By default, eBPF programs are JIT-compiled into native machine code before being executed, as the eBPF Just-in-Time flag is active by default in latest kernel releases.

B. Portability

The usage of hardware acceleration, including GPUs, to improve the performance of virtual functions has been largely investigated in the past, especially in the NFV domain. This approach provides an optimal migration path from pure hardware appliances to software instances, but also introduces placement and deployment constraints that jeopardize some of the main benefits of the cloud paradigm. As a matter of fact, traditional "centralized" security appliances (e.g., global datacenter firewalls) are hard to scale, leading to a more distributed approach in which this function is implemented directly on end hosts (e.g., datacenter servers).

To overcome most of the performance limitations that exist for software packet processing, kernel by-pass approaches are commonly used in NVF frameworks today. Despite of the large performance improvement, the main drawback with this approach is that all the network stack must be re-written; moreover, custom or modified versions of NIC drivers are usually required. Unfortunately, maintaining separate implementations from the standard kernel may become cumbersome and require a non-negligible maintenance cost. In addition, the availability and effectiveness of such technologies in a virtualized environment is questionable, especially in the case of public infrastructures and serverless models (i.e., Docker, Kubernetes). Indeed, the current market trends show an increasing interest towards the "cloud-native" model, where elementary functions are packaged in containers, deployed as microservices, and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows.

Differently from the kernel by-pass approach, we investigated the possibility to directly instrument the vanilla kernel. The introduction of the eBPF technology has overcome many typical limitations of packet inspection in the kernel, which has always proved hard to evolve and to program. By leveraging eBPF, which is a native kernel virtual machine available in all recent versions (4.19 and 5.x) commonly used in all installations, we achieve portability of our inspection tools across different infrastructures and cloud models. By extending the cloud workload protection model, we don't rely on any monitoring facility provided by the virtualization infrastructure, because the same virtual application and service already embeds the necessary instrumentation. By using eBPF, we do not either require additional kernel modules or features to be available, and we can use it in virtual machines as well as in Linux containers[7]. In addition, the current implementation uses Polycube to load eBPF programs and collect data. The specific Polycube function, called *dynmon*, can be dynamically loaded and remotely controlled by a REST-based interface, which simplifies the integration with ASTRID control plane (i.e., the LCP). Hence, we are effectively able to easily update the existing application by providing a replacement that does not disrupt the typical service workflow.

Definitely, our approach enables modern (cloud-native) applications to be easily deployed in traditional data-centers, in the cloud, in edge installations, and even in fog/IoT environments, with predictable performance. This effectively realizes the ASTRID objective of decoupling security processes from service management, because we don't put hardware or software constraints to the orchestration logic.

C. Extensibility

Fixed kernel network applications (or the associated kernel modules [9]) are notoriously slow and inefficient given their generality. In addition they have also proven hard to evolve due the complexity of the code, and the criticality to write safe code that does not harm the system in terms of stability, availability, performance, and data loss.

Traditionally, custom packet inspection tools have relied on raw sockets and the *pcap* library[8]. These are effective mechanisms to get raw packets in parallel to the Kernel, but they basically duplicate packets within the system and are largely ineffective to perform mitigation actions (like dropping or redirection).

By contrast, eBPF programs follow the same development process of userspace applications and can be created independently from the kernel development process. They are dynamically injected into the kernel, hence they allow the creation of inspection tasks that are tailored to the actual requirements of external detection and analytics processes, as opposed to static kernel implementations. The kernel provides a run-time verifier that analyzes all possible execution branches of an eBPF program before loading it. eBPF programs are rather limited in terms of memory access and available functions; indeed, operations outside the eBPF environment are carried out only through specific "helper" functions that further limit the likelihood of introducing faulty programs.

One major limitation for using eBPF programs is that they often require a specific userland counterpart to take care of control and management. For example, Suricata is currently able to load custom eBPF programs, but they are limited to a few operations (filtering, bypass, load balancing) and must comply with specific programming patterns[9].

In ASTRID, we addressed this issue by introducing *dynmon*, a transparent service that allows the dynamic injection of eBPF code in the Linux kernel, enabling the monitoring of the network traffic and the collection and exportation of custom metrics. The possibility to run custom code brings unprecedented flexibility in defining inspection rules, which are no more bounded to pre-defined patterns, as it happens in signature-based detection tools as Suricata, or events, as in Zeek, and are also more efficient than regular expressions or similar logic. Both Suricata and Zeek give access to a large number of protocol fields[10]; the Zeek script language is far more powerful than Suricata rules, but they are interpreted at run-time and not suitable for high packet rates. According to

[7]The scope of our approach does not cover unikernels. However, the interest in containers has largely shadowed these technologies, which are not anyway supported in a straightforward way by kernel-bypass frameworks either.

[8]https://www.tcpdump.org/.

[9]https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html#setup-ebpf-bypass.

[10]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.

the documentation, Zeek is not able to efficiently inspect fields that are present in each individual packet (e.g., IP headers).

The remarkable aspects of our approach is that exactly the same userland utility (namely, the dynmon service) is used to load and run programs that generate different metrics, without specific constraints on the data structure, even though structs and unions are not supported. This is mostly due to the export formats (JSON and OpenMetrics), because OpenMetrics does not support complex data structures. A large number of eBPF map types is currently supported, including plain and per-cpu hash, LRU hash, array, queue and stack. Finally, it also supports atomic eBPF maps content read thanks to an advanced map swap technique, and maps content deletion when read. The dynmon programming model is looser than what required by Suricata, allowing a larger degree of extensibility (even if the scope of the two tools is rather different).

Finally, we point out that extensibility does not necessarily mean that users have to re-develop everything from scratch. Since we introduced a new technology in ASTRID, the current platform only covers a few challenging use cases (DNS/NTP amplification attacks, network covert channels), but lack all common rules that are usually available in commercial and open-source projects. In this sense, it is not comparable with the maturity of direct competitors. This is due to the low TRL of the Project, that mostly focused on the introduction of new ground-breaking technologies rather than mere extensions of existing tools. Commercial exploitation would require a rich set of eBPF programs to cope with all common detection needs.

III. Polycube and dynmon

Polycube is conceived as a framework for developers of network functions in the NFV world [6]. Its peculiarity is the usage of eBPF for in-kernel packet processing, which is then combined with userland applications to allow both fast and slow paths, similar to what happens in the OpenFlow implementation. Polycube implements a control and management plane for eBPF programs, taking care of common operations as loading, configuration, data collection, unloading. It enables the creation of arbitrary and complex network function chains, with strong characteristics of isolation, persistence (e.g., across server reboots) and composability.

The elementary management unit is the *Cube*, which implements a specific type of network function: bridge, router, nat, load balancer, firewall, DDoS mitigator, etc. Cubes are similar to plugins that can be installed and launched at runtime. They plays a similar role to software images in common virtualization models, which can be used to create multiple instances with their own configuration, attachment points, and life-cycle management.

Several cubes can be chained together to create complex applications; the framework already provides a CNI plug-in for Kubernetes, which can handle the network of an entire data center, and a more efficient and scalable clone of the existing Linux iptables. This motivated the usage of Polycube to implement fast and lightweight security agents, which can therefore easily be integrated both in virtual network services (for instance, 5G deployments) and cloud-native applications.

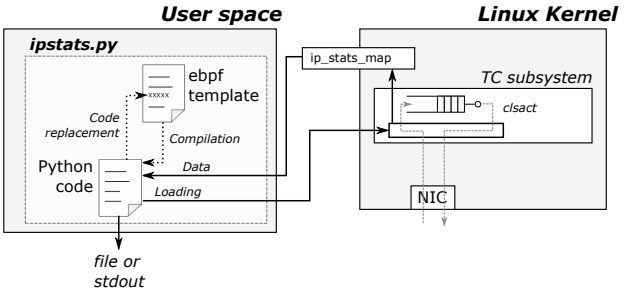


Fig. 1. Architecture of the bccstego framework.

An additional cube for monitoring network packets have therefore been developed, named *dynmon*. According to the Polycube architecture, it is composed of a data plane for packet inspection and a control plane for collecting and exporting measurements from the data plane. As already discussed in Section II, the innovative aspects of dynmon is the generic control plane designed to run heterogeneous eBPF programs, with loose a-priory limitations on the kind and structure of data produced. Eventually, this allows to collect any kind of measurements on the network traffic, only limited by the constraints on eBPF programs and data exporters[11].

IV. bccstego

bccstego was conceived as part of the CNR exploitation plans as standalone tool to run the same programs already developed for Polycube. The underpinning concept behind bccstego is automatic code generation at run time, which eliminates the need for developing and maintaining many different programs for similar purposes. This is clearly evident for both the amplification attacks and network covert channels described in Section V, where several slightly different programs have been developed for each use case. Indeed, as the name suggests, bccstego targets steganographic threats, which are challenging to address with a single program because of the virtually unlimited number of techniques that can be used. This represents a perfect scenario for run-time code generation, hopefully driven by some form on artificial intelligence in the future.

Fig. IV shows the current architecture of the framework. This is only the first kernel of our future work, so the inner logic for code generation is rather simple. Basically, the main executable `ipstats.py` is a Python application that follows the typical BCC framework[12]. A template for the eBPF program is therefore embedded into the same Python code; this template contains all necessary instructions for parsing IPv4/6, TCP and UDP protocol headers, and includes some placeholders. At run-time, according to the request for monitoring a specific field, the placeholders are replaced with appropriate code snippets, to read the current value of the field and to create the statistics described in Section V-B.

[11]The limitation on allowed export formats can be easily overcome if support for OpenMetrics is not explicitly required.

[12]BPF Compiler Collection, available on line: https://github.com/iovisor/bcc. Last Accessed: July 2021.

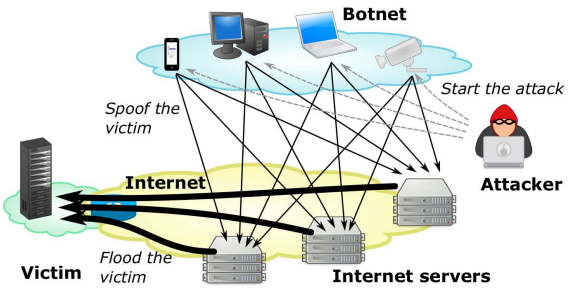


Fig. 2. Typical attack pattern for volumetric DDoS with amplification.

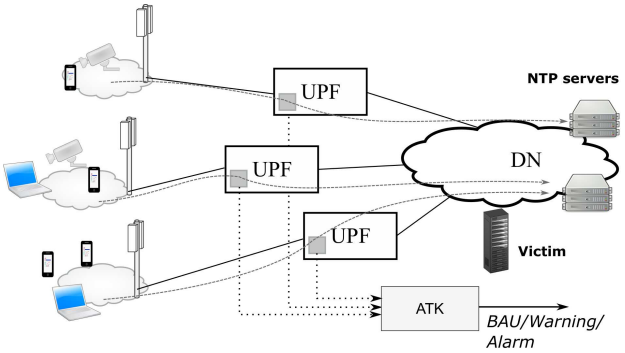


Fig. 3. The use case for NTP amplification attack.

Although it looks rather trivial, the current mechanism for generating the code requires minimal additional input to monitor a new field, hence facilitating the maintenance and extension to cover additional protocols. Indeed, all separate programs developed for Polycube are now generated from a common template in this tool. Of course, as already discussed in a companion report [7], the lightweight nature of eBPF programs require a more structured approach to parse many protocol layers.

V. Use case programs

Several eBPF programs have been developed in ASTRID based on multiple use cases, as briefly described in this Section.

A. Amplification attacks

Detecting amplification attacks is a challenging task, because there are multiple vulnerable protocols and thousands of servers scattered across the whole Internet [10]. From the victim perspective, detection of these attacks is trivial, because they fall under the umbrella of volumetric DoS (see Fig. 2); however, mitigation may be difficult, since the traffic comes from legitimate servers on the Internet. In any case, mitigating at the victim site is usually ineffective, because the Internet pipe is already saturated, not to count the overhead for exploited servers and the Internet itself.

Riding the NFV wave, in ASTRID we investigated the possibility to detect and mitigate amplification attacks originated in virtual instances of 5G networks [10]. The design of the 5G core follows a service-oriented architecture that facilitates its virtualization with different cloud models. Accordingly, it represents the ideal target for application of the ASTRID framework.

The description of the overall detection pipeline was already given in a previous paper [10]. Briefly, we integrate ASTRID agents in the User-Plane Function (UPF), and look for packets that may trigger an amplification attack, with the objective to stop it at its source (see Fig. 3).

Here, we only focus on the eBPF programs that have been developed for packet inspection in dynmon. We took into consideration two protocols, namely DNS and NTP, and their vulnerabilities. The NTP attack is based on sending a command called *monlist* to an NTP server; the server returns the addresses of up to the last 600 machines that it has interacted with. The request packet is only 234 bytes long, but the response may sum up to several dozens of kilobytes, depending on the number of returned addresses[13]. Similarly, the DNS attack maximizes the size of the response packets by using a query type of *ANY*, which means all available resource records should be returned[14].

We developed three distinct eBPF programs for each protocol:

- the first only measures the total number of query packets seen for that protocol;
- the second looks for the specific amplification pattern in the packet body, hence also giving the number of packets containing the *monlist* command (for NTP) or *ANY* query (for DNS);
- the last program is a modified version of the second, and discards packets containing the *monlist* command (for NTP) or *ANY* query (for DNS).

The source code of these programs is available in the ASTRID software repository[15].

B. Network covert channels

The detection of network covert channels is challenging, because there is no knowledge of which fields may be affected. However, the scope is usually restricted by the fact that many header fields cannot be changed without breaking operation of the corresponding protocol, hence triggering errors that can be easily detected.

In ASTRID, we developed a set of eBPF programs that are able to inspect the header fields of IP, TCP and UDP which could likely be used to implement a covert channel. The result of the inspection is an histogram that gives the number of occurrences of all possible values that a field can assume. The histogram can then be used to detect anomalies in the usage of the specific field; currently, a threshold-based mechanism has been developed by SIMARGL, but the idea is to make use of some form of machine learning [4].

To make the system more scalable, the full range of possible values is split into a number of equally-spaced bins; the number of bins can be selected to find the best balance between

[13] https://blog.cloudflare.com/understanding-and-mitigating-ntp-based-ddos-attacks/.
[14] https://www.cloudflare.com/learning/ddos/dns-amplification-ddos-attack.
[15] https://github.com/astrid-project/astrid-framework/tree/main/ebpf-programs/amplification.

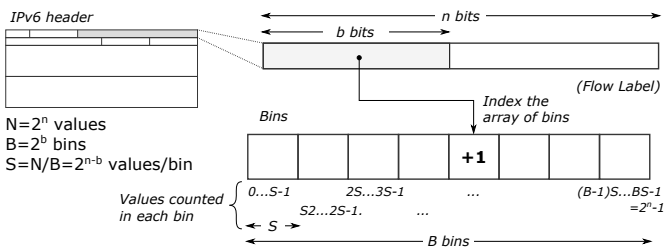


Fig. 4. Mapping field values to bins.

TABLE I
CONFIGURATION OF OPENSTACK SERVERS USED FOR TESTING.

Node	vCores	vRAM	vStorage
Sender	1	1 GB	16 GB
Receiver	1	1 GB	16 GB
Forwarder	4	2 GB	32 GB

precision and memory usage. Fig. 4 gives an example of operation of our eBPF programs for the flow label field of IPv6.

The current set of programs covers the following protocols and corresponding fields, which include almost every realistic implementation of covert channels at the network and transport layer (with the notably exception of ICMP):

1) IPv6: flow label, traffic class, hop limit [11];
2) IPv4: type of service/differentiated service code pointer, identification [12]–[14], time-to-live [15], [16], fragment offset [13], [17], internet header length;
3) TCP: ack number [14], [18], reserved bits [19], timestamp [20];
4) UDP: checksum [21]–[23].

All the programs are included in the ASTRID software repository[16]. They are also included in the standalone tool bccstego (see Section IV).

VI. EVALUATION

To demonstrate the claimed innovation achieved in the Project, our experimental evaluation was devoted to performance measurements and comparison with a legacy tool, namely Zeek. We considered both the impact of inspection tasks on the traffic (e.g., delay introduced in packet flows) and resource consumption in terms of CPU/memory usage.

We used three nodes in our testbed: the Sender that generates traffic, the Forwarder that runs the inspection agent, and the Receiver that is the final destination of the traffic. All nodes ran on the same hypervisor of an OpenStack installation, 2x Intel Xeon CPU E5-2660 v4@2.00GHz with 14 cores and hyperthreading enabled, 128 GB RAM, 64 GB SSD storage. The configuration of the OpenStack servers is reported in Table I. To make a fair comparison in a meaningful scenario, we ran our agents in a Docker container hosted in the Forwarder environment, which monitor the ingress interface.

We used two Traffic Generators (TG) in our experiments:

[16]https://github.com/astrid-project/astrid-framework/tree/main/ebpf-programs/stego.

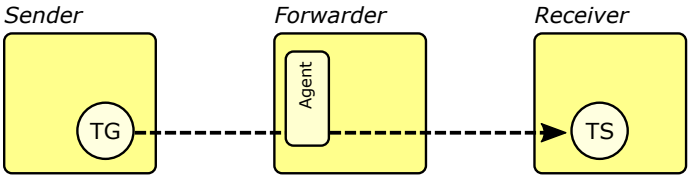


Fig. 5. The topology of the experimental testbed. All main blocks were Virtual Machines; the agent ran in a Docker container.

- *iperf*[17] was used to create IP traffic and to measure relevant performance indexes at the receiver side (packet rate, loss percentage, jitter);
- *tcpreplay*[18] was used to replicate real word packets.

At the Receiver side, we only used *iperf* whenever possible as Traffic Sink (TS).

Experiments were run for 10 minutes each, so to mitigate as much as possible interference with other parallel activity, both in the network and within the guest/host systems.

A. Zeek extensions

To evaluate the efficiency of our implementation, we considered Zeek. The choice was motivated by the fact that this tool performs a similar role than ASTRID agents, namely it provides raw measurements and delegates the detection to external processes. Zeek parses network packets and generates events, which are then processed by a powerful scripting language. The output from a Zeek script can be recorded in log files or can generate alerts for other cyber-security appliances.

Events generated by Zeek are mostly connection-oriented. For instance, for the DNS case, an event is generated at each request/reply, and contains relevant fields extracted from the message. Hence, the resulting script is rather simple: it just counts the number of DNS request events and the number of occurrences with a query type of *ANY*. There is a single program in this case because the event generated at the reception of a DNS query always contains the query type. The script and the configuration file to create the Docker container used in the experiments are also available through the ASTRID repository[19]. We do not expect meaningful differences in the behavior of Zeek in case of NTP messages, therefore we did not replicate the experiments for this protocol.

However, Zeek does not generate events on the reception of individual IP packets, for performance reasons. Therefore, to provide the same kind of functionality as our eBPF programs for network covert channels, we patched the source code and created a custom version that is able to report some fields from the IPv4/v6 and TCP headers. Once the event is available, we use the scripting language to catch it and to create the same kind of histogram as eBPF programs. The source code of the path, script, and docker file are kept in an external private repository[20].

[17]https://iperf.fr/.

[18]https://tcpreplay.appneta.com/.

[19]https://github.com/astrid-project/astrid-framework/tree/main/test/zeek-dns.

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

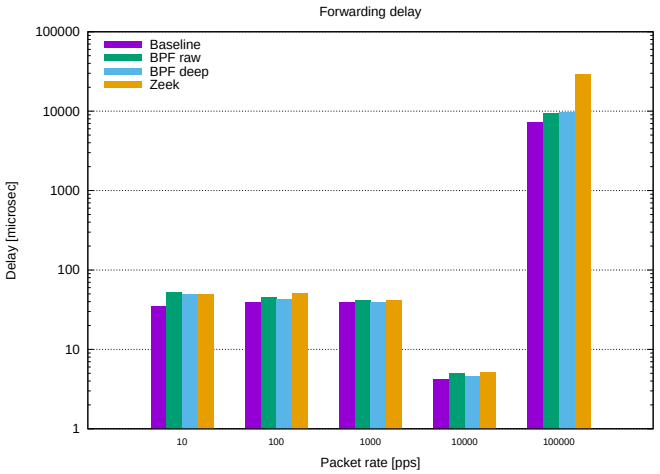


Fig. 6. Latency experienced by packets across the forwarding node.

B. DNS packet inspection

We evaluated the efficiency of our DNS packet inspection programs in Polycube. In this case, the Sender is representative of the attacker, the Forwarder node of the UPF in a 5G core, and the Receiver is the target for reflection. Indeed, our experimental testbed is a simplified version of the general scenario described in Sec. V, but it helps focus on the specific objectives for our evaluation. We didn't include any real 5G function in our experiments, because they are not relevant for our objectives. Finally, there is no server running at the receiver, again because this is not strictly necessary for our evaluation.

In these experiments we used *tcpreplay* to replicate two kinds of real-word DNS packets: a simple query for an A-type record, and a potentially rogue query of type ANY. To get meaningful results, each test was run for at least 30 seconds (tests with lower packet rate were run longer to get a larger number of samples). Our evaluation considered variable packet rates of 10, 100, 1000, 10,000 and 100,000 packets per second.

1) Impact on network traffic: The first evaluation consider the impact of the inspection agents on IP traffic. Our main finding was that we were able to receive packets at the full transmission rates, with no packet losses. We only observed 0.42% of packet lost for Zeek with the highest transmission rate. We therefore conclude that either of the inspection technologies does not affect packet transmission.

2) Forwarding delay: Even in the absence of packet losses, the deployment of an inspection probe may increase the packet processing time, which turns into more forwarding latency. This effect was not measurable in the previous experiment, because there is no perfect synchronization between the Sender and the Receiver (even if the accuracy with NTP is quite good). We therefore considered the overall latency experienced by packets across the Forwarder node. To this aim, we measured the difference in the timestamp recorder by *tcpdump* for each packet on the ingress and egress interface.

Fig. 6 compares the latency introduced by our programs and Zeek with a baseline scenario; the latter corresponds to the situation when no probe is applied. For our eBPF programs, "raw" indicates the simple packet counter and "deep" indicates the counter of packets with the ANY query.

With lower packet rates (namely, below 10,000 packets per second), the impact of the considered inspection agents is rather low. Indeed, even if the logarithmic scale largely hides the effect, the agents introduce up to 50% additional latency with respect to the baseline scenario. However, the overall latency is always below 100 μs, which is anyway more than acceptable for Internet traffic.

In some cases, Zeek seems to perform better than eBPF programs. However, while eBPF programs are really extensions to the operation of the in-kernel network stack, Zeek captures packets with a raw socket and then the inspection happens in parallel to forwarding operations implemented by the kernel. Overall, achieving close results to Zeek means that the impact on standard kernel operations is rather low. In addition, we remark that results may slightly vary for different realizations of the experiment, hence we can consider the performance almost equivalent.

Rather unexpectedly, the latency was lower for 10,000 packets per seconds. We repeated the experiments several times, and got the same results. One possible reason is better efficiency with this rate in the reception process (namely, fewer hardware interrupts are generated); in any case, since our objective is only to compare the tools, we did not investigate in detail the cause of this behavior.

With the largest packet rate, the gain with respect to Zeek largely increases. In this case, the latency introduced by Zeek almost doubles with respect to eBPF programs. Overall, the difference between raw and deep inspection is rather limited.

3) Processing delay: We already pointed out in the previous Section that eBPF programs are de-facto extensions to the operation of the kernel. So it is interesting to look at the time to execute such programs. Even if this measurement is not trivial to perform, we provide a reasonable estimation by measuring the elapsed time from within the same program (namely, timestamps are taken at the entry/exit point). Of course, this does not account for the overhead to invoke and run the eBPF function.

Fig. 7 shows the execution times measured under different conditions. This time, we also considered the potential impact of different queries, namely a request for an A record and an ANY request. This impact is practically negligible from our experiments; as in the previous case, the figures may slightly changes in different realizations. Instead, the deeper inspection requires more time, as expected, because of the larger number of operations to be performed. Overall, the processing time is always well below 1 μs, hence confirming the high efficiency of this inspection method.

Similarly to the previous experiment, shorter processing times are measured for the highest packet rates. Again, the same behavior was observed for different realizations and we do not have an explanation for this behavior; however, it is worth being investigated deeper in the future.

4) Resource usage: Finally, we consider resource usage, by using again the setup with *tcpreplay*, and investigate disk usage, memory allocation and CPU usage.

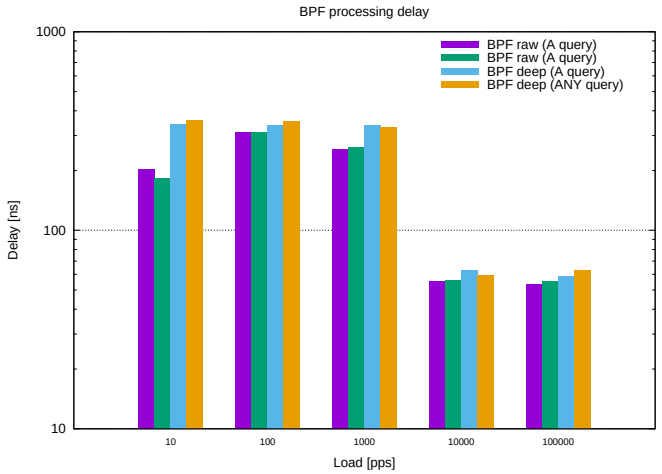


Fig. 7. Execution times for eBPF programs, when processing A and ANY queries.

TABLE II
COMPARISON BETWEEN DOCKER IMAGE SIZES.

Tool	Base	Additional	Total
zeek-dns	124MB (Debian testing)	307MB	431MB
polycube	63.1MB (Ubuntu 18.04)	206MB	269MB

Table II compares the size of docker images used in the experiments. We roughly broke down the total size into the base image size and the total of additional layers added by the tool. The zeek image is larger; even if we consider that we started from a bigger base image, the Zeek installation took around 100MB more than Polycube[21]. This is not surprisingly, because Zeek is a powerful flow processor tool; unfortunately, most features are always embedded and cannot be disable when simper operations are required. We also note that the Polycube image includes all the cubes included in the framework; hence, the additional layer to run dynmon is practically negligible. For what concerns eBPF programs, they are injected at run-time, hence not included in the figures of Table II; however, their contribution is of a few kilobytes and can be neglected.

Fig. 8 shows global CPU usage in both user-space (right side) and kernel-space (left side). In this case, in addition to packet rates used in other experiments, we also included the fastest rate tcpreplay can send packets (which is basically the same that can be achieved with 100,000 pps). Even if the figures cannot be ascribed to the monitoring tools only, it is clear that the impact of Zeek on CPU usage is more than an order of magniture greater than Polycube. In the case of Polycube, since all inspection operations are done in kernel space, we note an increment of system CPU usage with higher transmission rates, whereas there are no meaningful changes for the user space, which only collects data with the same rate. Instead, Zeek splits operation between kernel-space (for capturing and duplicating packets) and user-space (for packet inspection), and this is reflected in the increment of both system and user CPU usage.

The total computing overhead of Zeek is even more evident when we look at the cumulative CPU usage. Fig. 9, which does not use the logarithmic scale, shows how the CPU usage of Zeek rises up from 10,000 packets onward, which are the likelihood conditions for a large-scale attack.

Finally, Fig. 10 shows an analysis of memory allocation for the different tools. Not surprisingly, the virtual memory size (VMS) of Zeek is far larger than Polycube, because the program is in general more complex. It is also rather intuitive that the eBPF programs does not impact memory allocation for Polycube. For each tool, the RSS and PSS are basically the same, because libraries are not shared with other processes inside the docker container. Also the Resident Set Size (RSS) and Proportional Set Size (PSS) of Zeek are higher than Polycube, even if proportionally less than the VMS. If we compare Zeek with our previous experiments [7], we need less memory allocation in this case. This is because we are running a single script on one interface in this experiment, whereas in our previous work we ran the full stack on two interfaces.

By considering the break down of memory usage for the RSS, we note that Zeek allocates less memory to the process itself and shared libraries ("proc" and "lib" slices, respectively), but have around the same heap size. Remarkably, a large slice of memory indicated as "socket" is used for capturing packets with the raw socket.

C. Inspection of IP/TCP/UDP protocol headers

We already investigated the precision of the data and its usage in our previous work [3], [4], therefore in this study we only focus on the impact on network operation and resource usage. Accordingly, we do not need to reproduce covert channels in our testbed, but just to replicate several traffic conditions, also considering different transport protocols. For this reason, we used *iperf* to investigate the influence of the following parameters on the performance of the detectors:

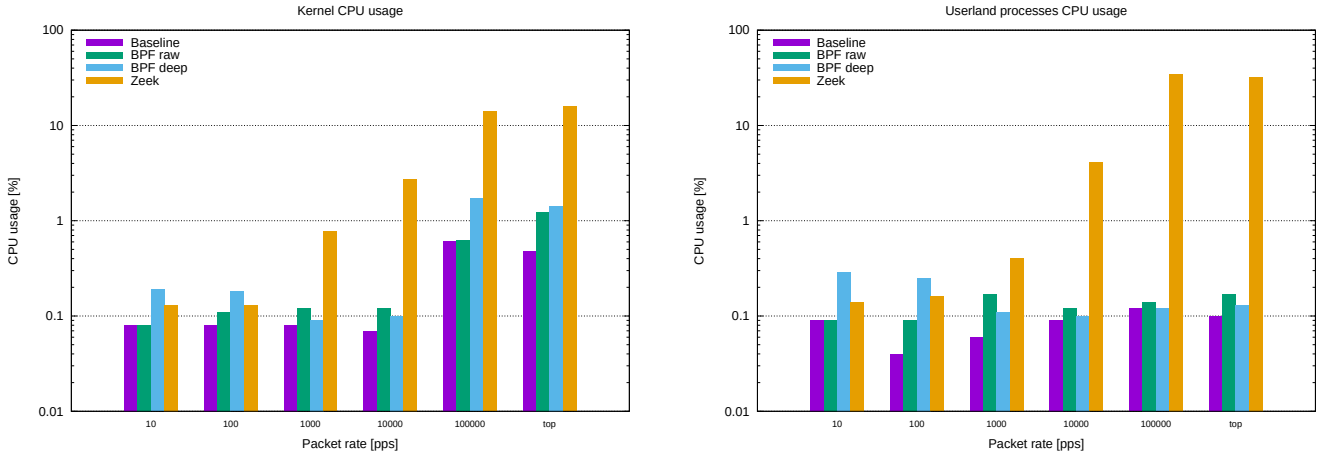
- packet size and transmission rate for UDP flows;
- maximum segment size (MSS) for TCP streams.

For packet size, we considered 4 values that are representative of the following cases:

- 16 bytes is the smallest value allowed by iperf, and this is the worst condition for packet forwarding;
- 1470 bytes is representative of the biggest Ethernet packets, also accounting for the presence of tunneling in the underlying virtual network;
- 8192 bytes is the reference size for jumbo frames in Ethernet, which is a common situation in all installations;
- 65507 bytes is the maximum size allowed by UDP and the best condition for packet forwarding, but it is only feasible on loopback interfaces (therefore, it can only be used when VMs are running on the same host).

For the transmission rate, we considered a broad range of different load conditions, from 10 Kbps to the unfeasible (at least for our installation) rate of 10 Gbps. For the MSS, we again selected 4 values that this time are representative of: the

[21]The Zeek image includes zeek executables and the necessary libraries and tools.



(a) Kernel (b) User

Fig. 8. CPU usage measured at the intermediate node, while varying the transmission rate of DNS packets.

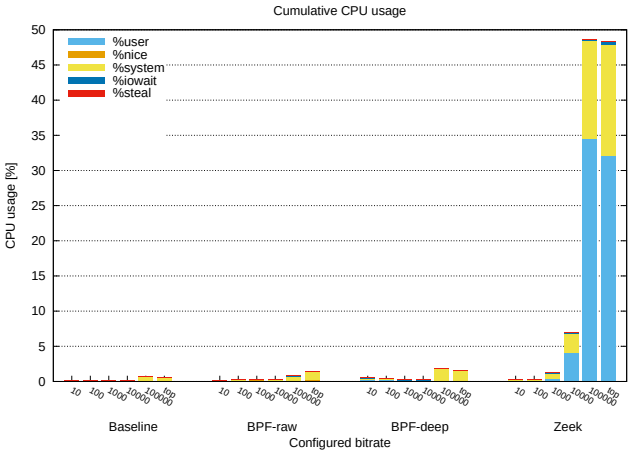


Fig. 9. Cumulative CPU usage measured at the intermediate node, while varying the transmission rate of DNS packets.

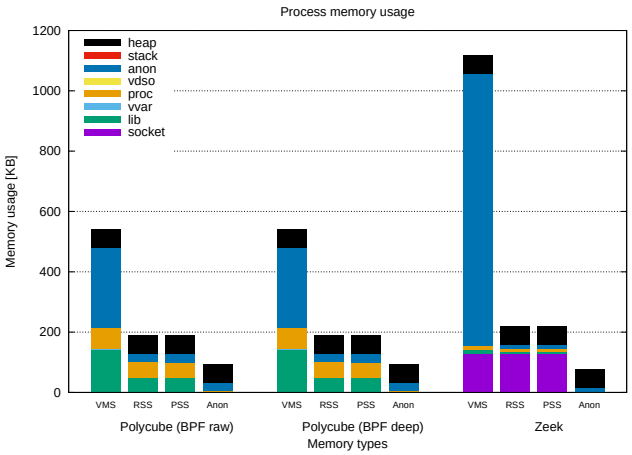


Fig. 10. Memory allocation for the different tools.

smallest value accepted by iperf3 (88 bytes), the minimum value that should be used on IP links (536 bytes), the typical value used for Ethernet links (1460 bytes) and the maximum value accepted by iperf3 (9216 bytes).

The comparison includes the "baseline" scenario (where no tool runs, just to understand the upper bounds on the experimental setup), the standalone tool bccstego, the patched Zeek version (that analyzes per-packet events as described in Sec. VI-A), and a simple implementation of our inspection mechanism in C (based on pcap). The last tool represents the simpler form of parsing we can do to inspect network packets, by leveraging the pcap framework. It is expected to provide a lower bound to memory allocation and CPU usage in user space, since compiled C is much more efficient than interpreted Python code of bccstego. Kernel by-pass modules were not considered because they are largely ineffective in our setup [7].

We also compared the results got for monitoring different fields, in order to verify the impact of parsing different protocols and using more or less bins. Here, we only report the results for the Flow Label (IPv6) and Time-To-Live (IPv4) which uses 2^{12} and 2^{8} intervals, respectively. Additional results that consider other protocols and fields can be found in a companion paper recently submitted [24].

1) Impact on packet transmission: We evaluate the impact on packet transmission by considering the measurements reported by iperf3, namely the transmission rate, packet error rate and jitter. Fig. 11 shows how the measured bitrate at the receiver changes for different settings of packet size and transmission bitrate for UDP flows. With smaller packet size, it is not possible to achieve the higher bitrates, and this is a general understanding for Ethernet links.

We note that all probing tools have limited impact with respect to the transmission rate. Pcap-based tools duplicate packets at the raw socket layer, hence the additional processing have not a direct impact on forwarding operations. However, eBPF programs are invoked on the kernel path, so it is a very good result that our implementation does not limit the maximum bandwidth with respect to the baseline. Finally,

we did not notice significant differences when monitoring a different field and protocol or using a different number of bins.

Fig. 12 measures the percent packet loss at the receiver under the same conditions. As expected, it is correlated to the bitrate shown in Fig. 11. When the transmission cannot achieve the desired bitrate, we see higher packet loss. Apart the case of small packets, packets are mostly dropped at the higher bitrates (1/10 Gbps), which are not feasible in our setup. If we do not consider these scenarios, the degradation of the probing mechanisms with respect to the baseline is rather limited, also considering that the results slightly change in different realizations. Again, there is no significant differences when monitoring a different field/protocol or using a different number of bins.

The last UDP measurement is the average packet jitter, shown in Fig. 13. The variation in the inter-packet delay is negligible for practical applications (below 0.1 ms for all scenarios but the largest packet size), and it is likely to be highly affected by external factors (including perturbations in the generator and the receiver). This means that the impact on the transmission of real-time traffic (e.g., multimedia) would be largely negligible. Similarly to previous indexes, there is no tool which wins over all in every condition. Finally, no meaningful differences are present when monitoring different fields/protocols or using a different number of bins.

Finally, we show in Fig. 14 the transmission rate achievable by TCP when generating packets for the whole duration of the experiment. Not surprisingly, higher bitrate are possible with larger MSS, because this has a beneficial impact on the TCP flow control mechanism.

By looking at the performance indexes reported in this Section, we can conclude that our eBPF-based mechanism does not affect packet transmission in a significant way.

2) CPU usage: CPU and memory usage are important to understand what is the impact on the operation of other applications. We expect significant differences in the usage of CPU between bccstego and the other tools, due to the different architectural design. As a matter of fact, Zeek and our pcap-based tool are basically user-space applications. The default capture driver for Zeek is libpcap[22]. On the other hand, our bccstego tool leverages in-kernel eBPF programs, hence the usage of CPU from kernel and userspace is expected to be rather different.

This is only partially confirmed by the measurements reported in Figg. 15 and 16 for a UDP flow, which show much higher CPU usage for user-space in case of our pcap tool and Zeek. Both tools increase CPU usage at higher bitrates, whereas bccstego is more constant across the different conditions. Differently from what expected, there is a non-negligible increase in kernel CPU usage for the pcap tool and Zeek; this is probably due to the mechanism implemented by libpcap to retrieve the packets. (please note the logarithm scale is used in this case).

A more detailed breakdown of CPU usage confirms that most time is spent in kernel and user-mode rather than other states. Fig. 17 shows that bccstego always brings a small overhead with respect to the baseline. Quite interesting, due to the implementation in Python, CPU usage is larger than Polycube (see Fig. 9).

Similar considerations hold in case of TCP, which measurements are shown in Fig. 18, 19, and 20. Even if the Python language is not the best option in terms of processing speed, bccstego performs better than all other alternatives, and limit the additional CPU usage with respect to the baseline.

3) Memory allocation: For each user-space application, we considered the memory size (virtual memory also accounting for external libraries), Resident Set Size (RSS, namely the actual size of physical memory, again including shared libraries), Proportional Set Size (PSS, namely the size of physical memory with proportional attribution of shared libraries), and Anonymous (the stack and other allocations not mapped to files). Fig. 21 shows that Zeek has a larger memory space, but only a minimal part is allocated in the RAM[23]. The memory allocated to bccstego is larger because of the many libraries needed by Python, which largely overcome what used by Zeek. Interestingly, we see the impact of the larger number of bins, reflected in the share of "anonim" memory.

The simple C implementation based on libpcap has a negligible memory footprint, because it only uses the minimal system libraries for input/output and libpcap. This suggests the possibility to rewrite bccstego in C and switch to libbpf[24] instead of BCC. Even if BCC hides the detail of compiling eBPF programs for the specific system where it will run, new developments promise the possibility to directly inject compiled code into the verifier[25].

VII. Related work

As typically happens for Kernel features (e.g, namespaces), the eBPF technology provides the main enabler for code augmentation, but control and management are left for third-party tools. For monitoring performance issues, this is largely implemented by the *BPF Compiler Collection* (BCC)[26]. Beyond the rich set of ready-to-use tools, this framework provides a python class that hides the technical details of compiling and loading eBPF programs; however, users are still required to develop their own code for retrieving and exporting data.

A higher-layer interface is provided by *bpftrace*[27], a tracing language that compiles scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system and existing Linux tracing capabilities. The bpftrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap.

[22]There are alternative capture drivers that can be used with Zeek, including raw sockets, libpcap, and PF_RING. We only considered libpcap in our study.

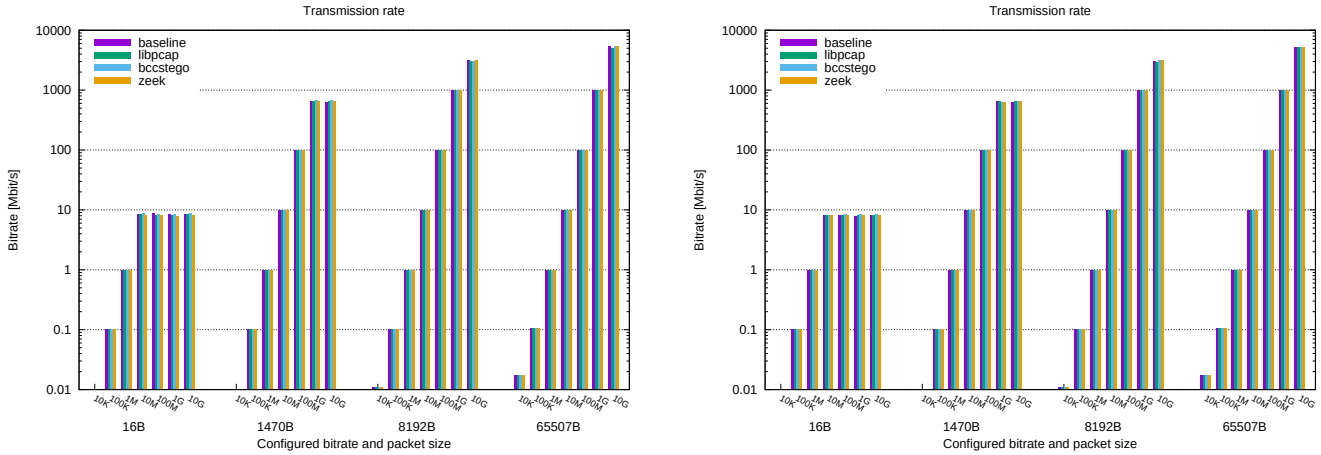
[23]There are some differences in the breakdown of memory allocation with respect to Fig. 10 (e.g., the "socket" share). This is because in case of amplification attacks we used the installation provided by Debian, whereas for investigating network covert channels we patched and therefore compiled our own version of Zeek, without replicating the same compilation flags as the Debian version.

[24]https://github.com/libbpf/.

[25]See BPF Portability and CO-RE, https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html.

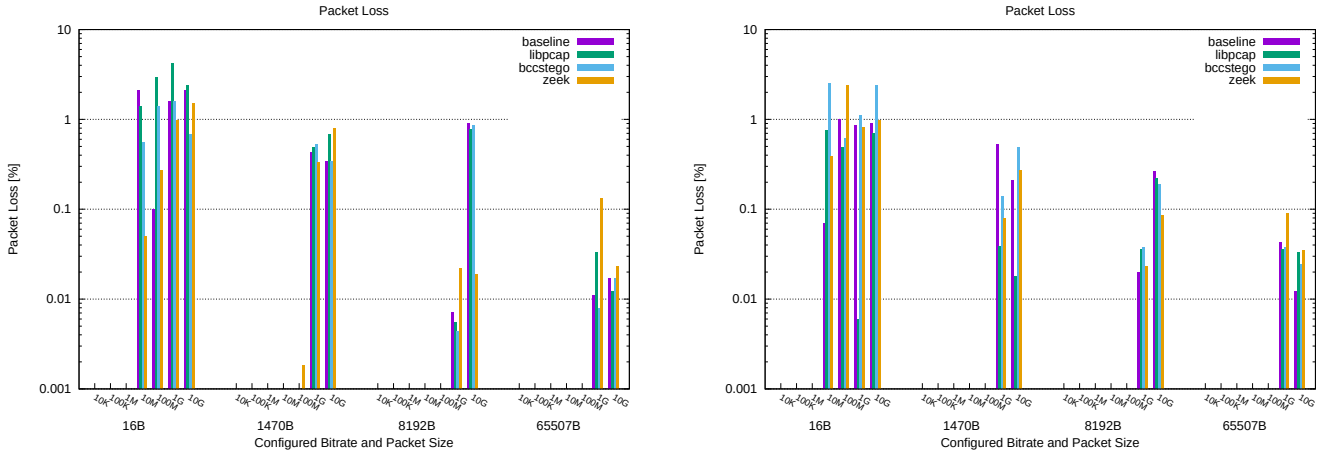
[26]https://github.com/iovisor/bcc.

[27]https://github.com/iovisor/bpftrace.



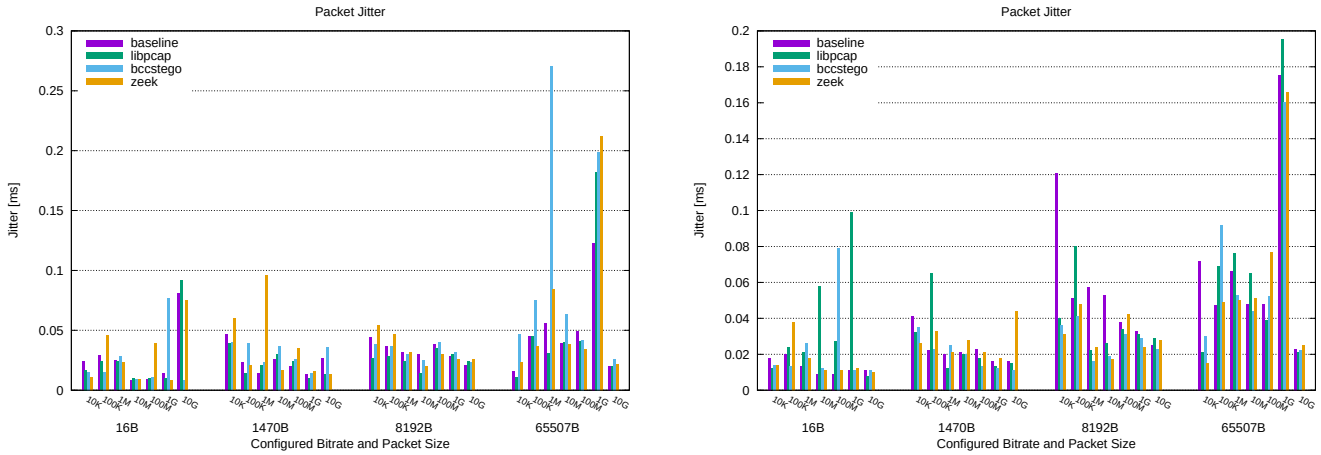
(a) Flow Label (b) Time-To-Live

Fig. 11. Measured bitrate at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



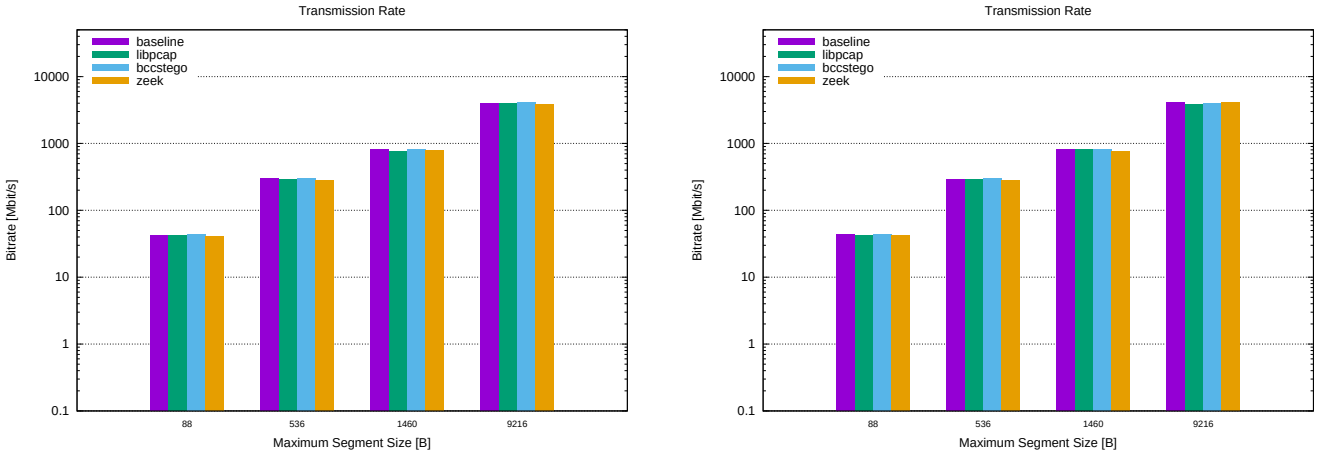
(a) Flow Label (b) Time-To-Live

Fig. 12. Measured packet loss at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



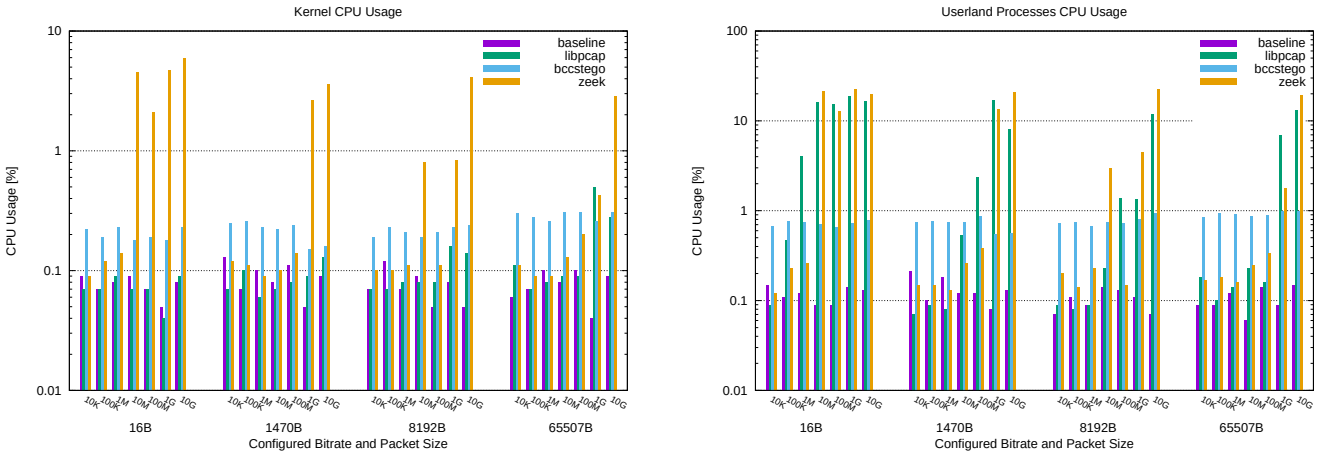
(a) Flow Label (b) Time-To-Live

Fig. 13. Measured packet jitter at the receiver, while varying packet size and the transmission bitrate for a UDP flow.



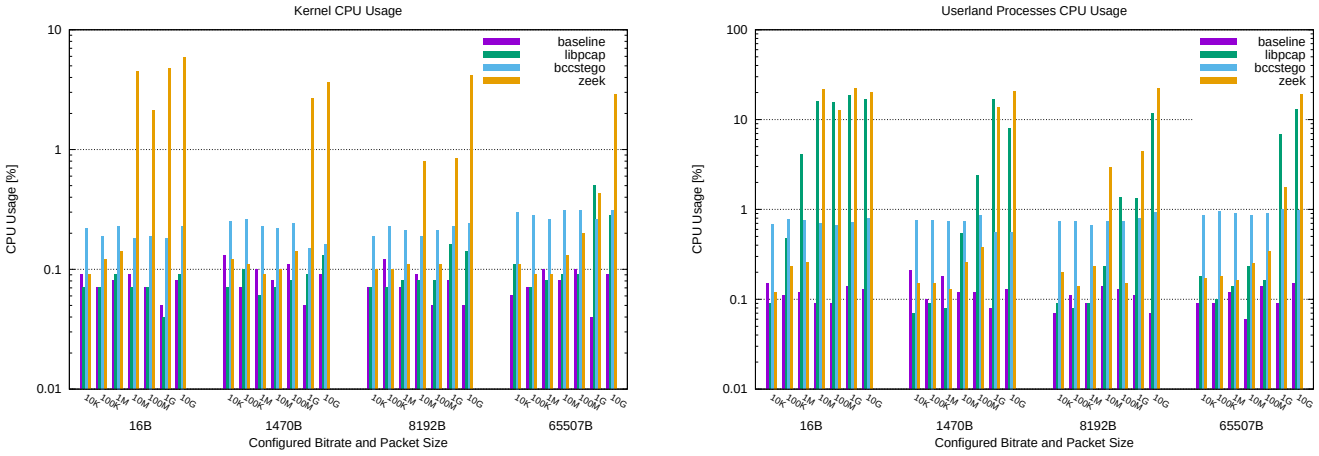
(a) Flow Label (b) Time-To-Live

Fig. 14. Measured bitrate at the receiver, while varying the MSS for a TCP flow.



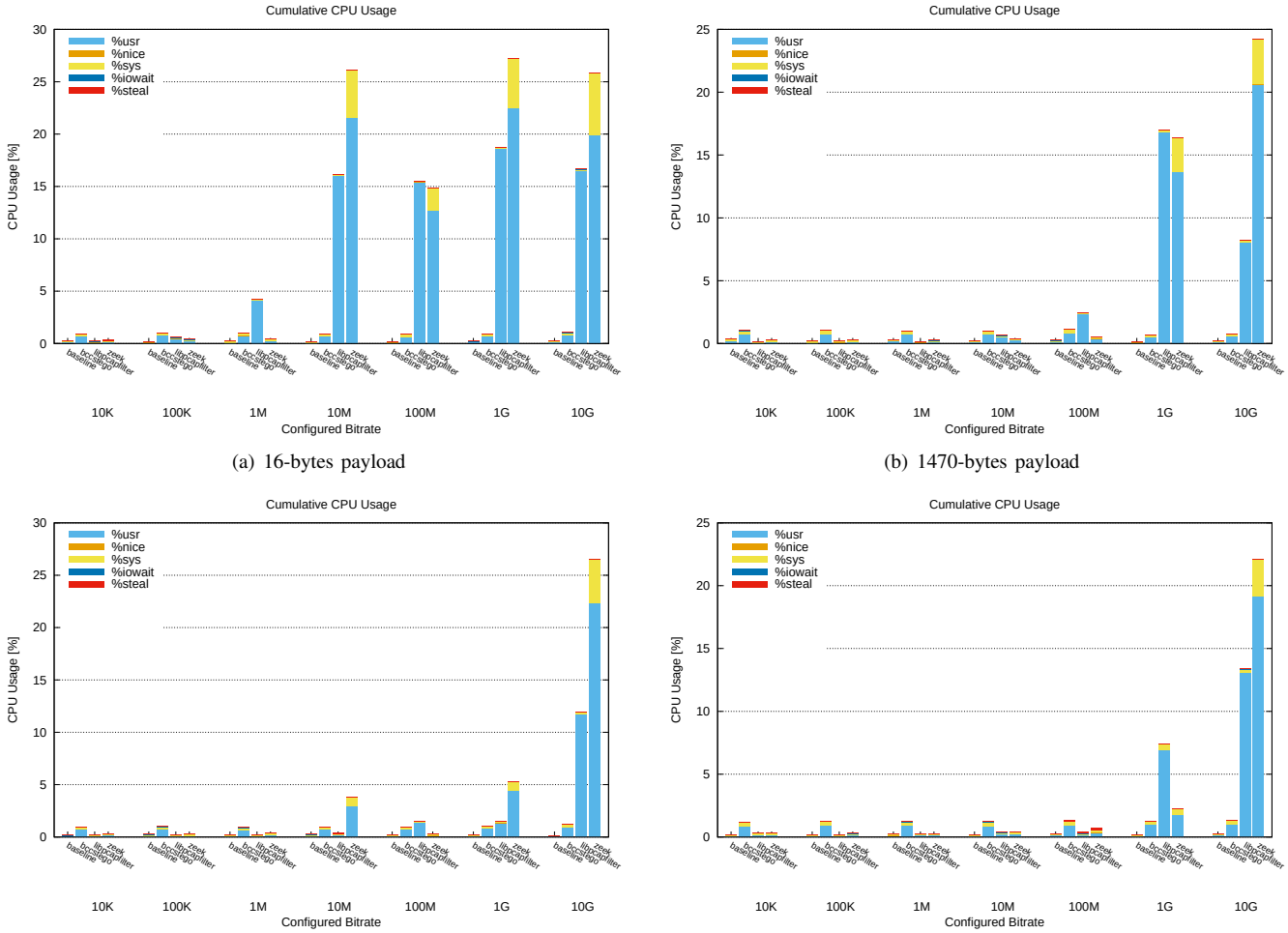
(a) Kernel (b) User

Fig. 15. CPU usage measured at the intermediate node, while varying the packet size and transmission bitrate for a UDP flow. The monitored field is Flow Label.



(a) Kernel (b) User

Fig. 16. CPU usage measured at the intermediate node, while varying the packet size and transmission bitrate for a UDP flow. The monitored field is Time-To-Live.



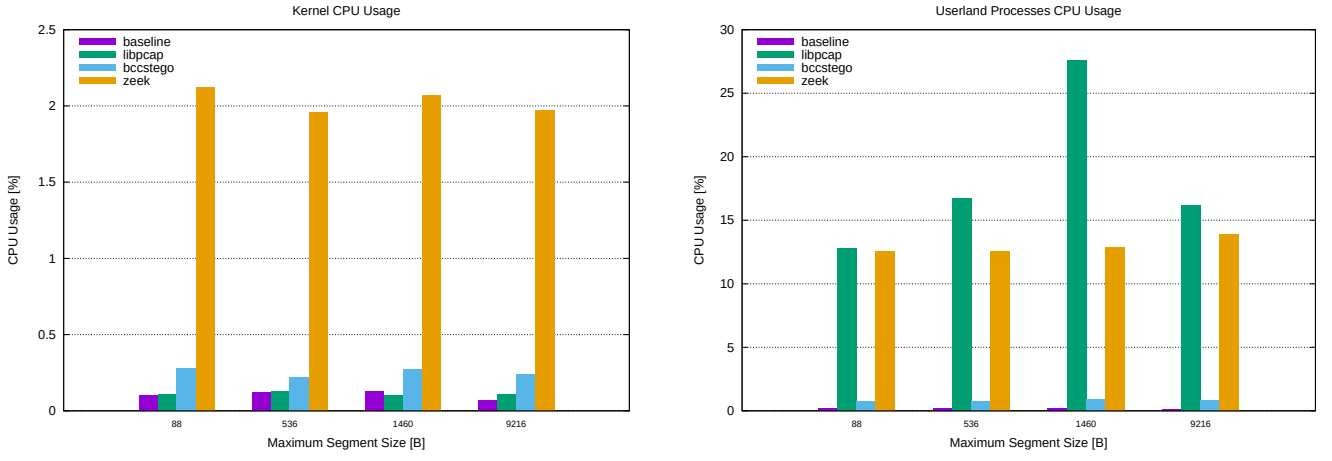
(a) 16-bytes payload

(b) 1470-bytes payload

(c) 8192-bytes payload

(d) 65507-bytes payload

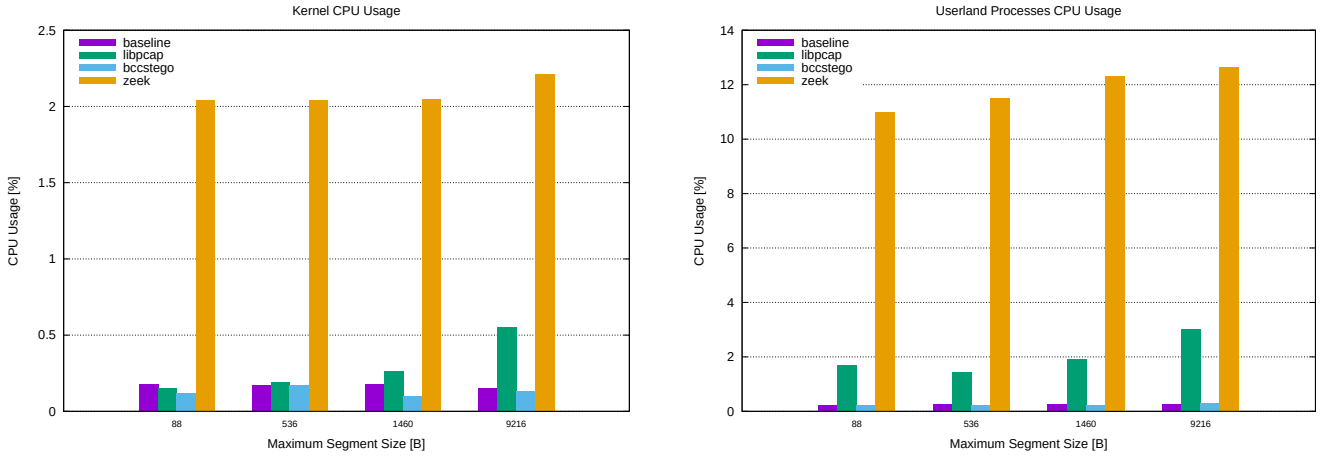
Fig. 17. Cumulative CPU usage measured at the intermediate node, for a UDP flow. The monitored field is the Flow Label.



(a) Kernel

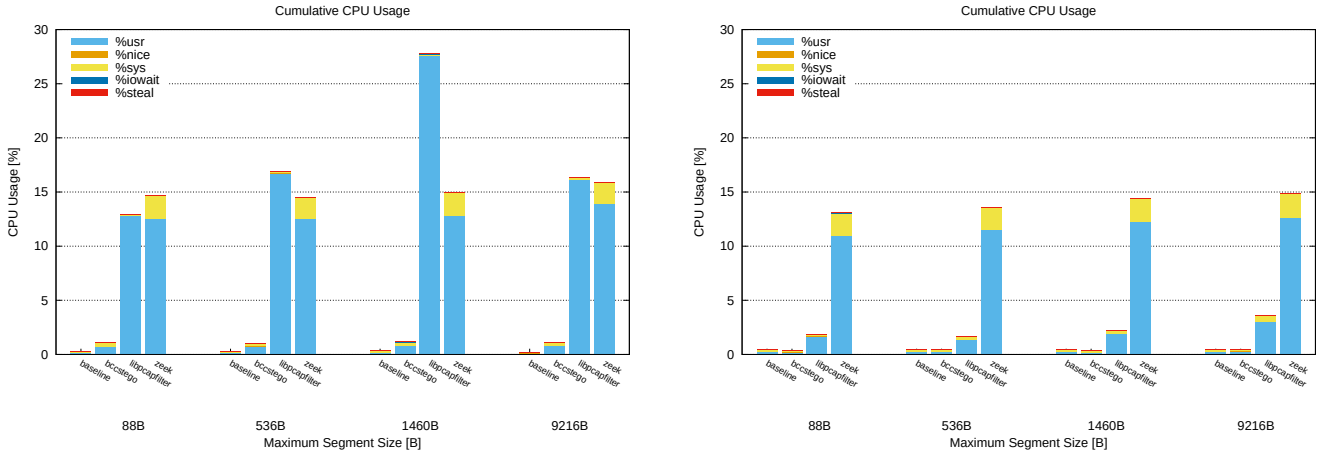
(b) User

Fig. 18. CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is Flow Label.



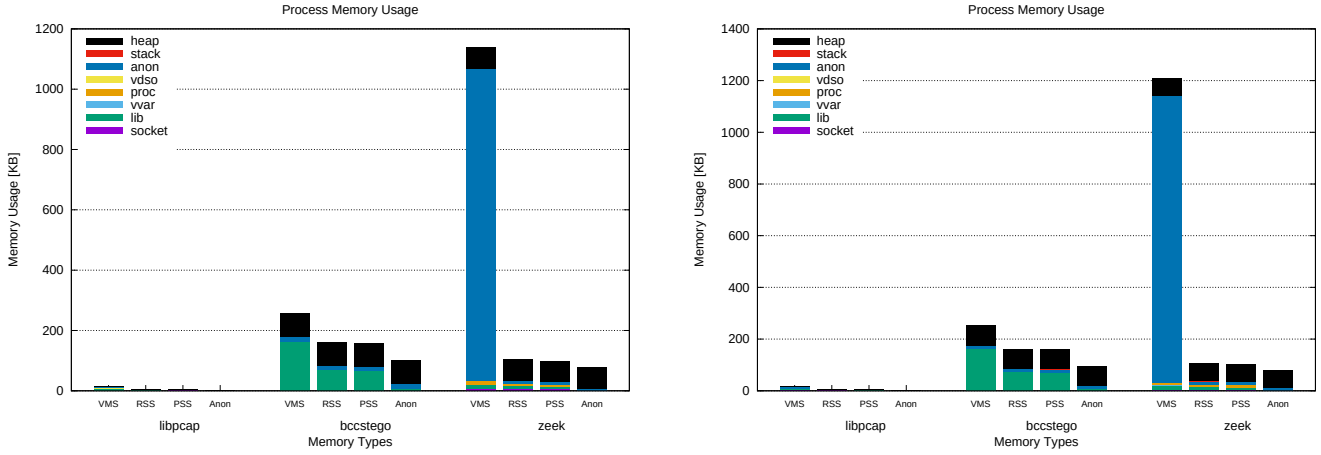
(a) Kernel (b) User

Fig. 19. CPU usage measured at the intermediate node, while varying the MSS for a TCP flow. The monitored field is Time-To-Live.



(a) Flow Label (b) Time-To-Live

Fig. 20. Cumulative CPU usage measured at the intermediate node, for a TCP flow.



(a) Flow Label (b) Time-To-Live

Fig. 21. Memory allocation for the different user-space tools.

VIII. Conclusion

In this paper, we have conducted detailed performance evaluation for a set of eBPF programs developed to cover the monitoring needs of ASTRID use cases. Our analysis took into consideration the programs themselves and the framework to dynamically load and manage them at run-time, namely Polycube. In addition, we included a first artefact which is part of our exploitation of ASTRID knowledge, namely bccstego. We demonstrated that our initial objective of having lightweight agents suitable to be integrated in virtualized services have been achieved. As a matter of fact, our framework have similar impact on packet transmission as other well-known tools, but far less CPU and memory usage.

We also discussed the greater flexibility brought by leveraging the eBPF framework. Simple programs can be developed to create custom statistics on any protocol field; other tools, as Zeek, have a powerful scripting language to build custom metrics as well, but they lack the necessary events for all the protocol stack. As a matter of fact, managing per-packet events with Zeek is not possible; custom extensions are required to use this feature, but they lead to great performance degradation with respect to our tools. The main drawback of our approach is the steeper learning curve than typical scripting languages and rule patterns; however, once the program structure and the restrictions imposed by the verifier have been understood, the flexibility and versatility of C-like programming is incomparable.

We are currently planning to demonstrate our approach with further challenging use cases; in addition, we are interested in pursuing the concept of dynamic code generation in a more structured way. Additionally, we plan to extend the comparison with other tools, mainly Suricata, at least for what concerns amplification attacks.

Acknowledgment

This work was supported in part by the European Commission under Grant Agreement no. 786922 (ASTRID).

References

[1] 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*, vol. 85, pp. 235–249, August 2018.

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

[3] 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*, vol. 191, May 2021.

[4] L. Caviglione, M. Zuppelli, W. Mazurczyk, A. Shaffhauser, and M. Repetto, "Code augmentation for detecting covert channels targeting the ipv6 flow label," in *IEEE International Conference on Network Softwarization*, Tokyo, Japan (Virtual), Jun., 28th –Jul., 2nd 2021.

[7] M. Repetto and A. Carrega, "Efficient flow monitoring for virtualized applications with ebpf," ASTRID project, Tech. Rep., July 2021. [Online]. Available: http://doi.org/10.5281/zenodo.5113889.

[5] S. Wendzel, S. Zander, B. Fechner, and C. Herdin, "Pattern-based survey and categorization of network covert channel techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–26, 2015.

[6] 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*, vol. 18, no. 1, pp. 133–151, March 2021.

[8] Netronome, "Avoid kernel-bypass in your network infrastructure," Blog post, January 2017. [Online]. Available: https://web.archive.org/save/https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/

[9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of openvswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA – USA, May, 4th–6th 2015, p. 117–130.

[10] M. Repetto, A. Carrega, G. Lamanna, J. Yusupov, O. Toscano, G. Bruno, M. Nuovo, and M. Cappelli, "Leveraging the 5g architecture to mitigate amplification attacks," in *IEEE International Conference on Network Softwarization*, Tokyo, Japan (Virtual), Jun., 28th –Jul., 2nd 2021.

[11] N. B. Lucena, G. Lewandowski, and S. J. Chapin, "Covert channels in ipv6," in *Proceedings of: the 5th International Workshop Privacy Enhancing Technologies (PET 2005)*, Cavtat, Croatia, May 30th–Jun. 1st 2005, p. 147–166.

[12] K. Ahsan, "Covert channel analysis and data hiding in tcp/ip," Master's thesis, University of Toronto, 2002.

[13] E. Cauich, R. Gómez, and R. Watanabe, "Data hiding in identification and offset ip fields," in *Proceedings of 5th International School and Symposium of Advanced Distributed Systems (ISSADS)*, ser. LNCS, F. F. Ramos, V. L. Rosillo, and H. Unger, Eds. Springer, 2005, vol. 3563, pp. 118–125.

[14] C. H. Rowland, "Covert channels in the tcp/ip protocol suite," DoIS Documents in Information Science, May 1997.

[15] H. Qu, P. Su, and D. Feng, "A typical noisy covert channel in the ip protocol," in *Proceedings of the 38th Annual International Carnahan Conference of Security Technology*, Albuquerque, NM – USA, Oct.11th–14th 2004, p. 189–192.

[16] S. Zander, G. Armitage, and P. Branch, "Covert channels in the ip time to live field," in *Proceedings of the Australian Telecommunication Networks and Applications Conference (ATNAC)*, Melbourne, Australia, Dec.4th–6th 2006.

[17] W. Mazurczyk and K. Szczypiorski, "Steganography in handling oversized ip packets," in *First International Workshop on Network Steganography (IWNS 2009)*, Wuhan, China, Nov. 18th–20th, 2009.

[18] X. Luo, E. Chan, and R. Chang, "CLACK: A network covert channel based on partial acknowledgement encoding," in *IEEE International Conference on Communications*, Dresden, Germany, Jun., 14th–18th 2009, pp. 1–5.

[19] P. Allix, "Covert channels analysis in tcp/ip networks," Master's thesis, IFIPS School of Engineering, University of Paris-Sud XI, Orsay, France, 2007.

[20] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts, "Covert messaging through tcp timestamps," in *Proceedings of the 2nd international conference on Privacy enhancing technologies (PET 2002)*, San Francisco, CA – USA, Apr., 14th–15th 2002, p. 194–208.

[21] C. Abad, "Ip checksum covert channels and selected hash collision," University of California, Tech. Rep., 2001.

[22] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, "Eliminating steganography in internet traffic with active wardens," in *5th International Workshop on Information Hiding (IH)*, ser. LNCS, F. A. P. Petitcolas, Ed., 2002, vol. 2578, pp. 18–35.

[23] J. S. Thyer, "Covert data storage channel using ip packet headers," SANS Institute, Tech. Rep., 2008.

[24] M. Zuppelli, A. Carrega, and M. Repetto, "An effective and efficient approach to improve visibility over network communications," Submitted to Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), August 2021.