bccstego: A Framework for Investigating Network Covert Channels

Matteo Repetto
matteo.repetto@ge.imati.cnr.it
Institute for Applied Mathematics and Information Technologies, CNR
Genova, Italy

Luca Caviglione
luca.caviglione@ge.imati.cnr.it
Institute for Applied Mathematics and Information Technologies, CNR
Genova, Italy

Marco Zuppelli
marco.zuppelli@ge.imati.cnr.it
Institute for Applied Mathematics and Information Technologies, CNR
Genova, Italy

ABSTRACT

Modern malware increasingly exploits information hiding to remain undetected while attacking. To this aim, network covert channels, i.e., hidden communication paths established within legitimate flows, can be used to exfiltrate data or exchange commands without getting noticed by firewalls, antivirus, and intrusion detection systems. Since the secret data can be directly injected in various portions of the stream or encoded via suitable alterations of the traffic, spotting hidden communications is a challenging and poorly generalizable task. Moreover, the majority of works addressed IPv4, thus leaving the detection of covert channels targeting IPv6 almost unexplored.

This paper presents `bccstego`, i.e., an inspection framework for computing statistical indicators to reveal covert channels targeting the IPv6 header. The proposed approach has been designed to be easily extended, for instance to search for channels not known a priori. Numerical results demonstrate the effectiveness of our first tool in the `bccstego` framework as well as its ability to handle high-throughput IPv6 flows without adding additional delays.

CCS CONCEPTS

• **Networks → Network performance evaluation**; • **Security and privacy → Network security**; *Domain-specific security and privacy architectures.*

KEYWORDS

stegomalware, network covert channel, packet inspection, eBPF, BPF compiler collection

ACM Reference Format:
Matteo Repetto, Luca Caviglione, and Marco Zuppelli. 2021. bccstego: A Framework for Investigating Network Covert Channels. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–05, 2021, Virtual Event*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465481.3470028

1 INTRODUCTION

In recent years, the Internet has become the target of sophisticated attacks causing huge economical losses. Threats like cryptolockers, ransomware, cryptominers and advanced persistent threats are endangering individuals and large-scale organizations on a daily basis, highlighting the limits of standard security tools and practices [1]. An emerging trend concerns the use of some form of information hiding or steganography to create malicious software which is difficult to detect, defined as *stegomalware* [2]. For instance, stegomalware can hide data within an innocent-looking carrier to prevent its detection, bypass blocks enforced by a firewall, implement stealthy multi-stage loading architectures, or escepe security perimeters implemented by sandboxes [1, 2]. Among the various information hiding techniques, the ability of creating network covert channels is gaining popularity [1–3]. In essence, a network covert channel enables two remote endpoints to secretly communicate by injecting data within a legitimate traffic flow. This allows to remain under the radar while orchestrating a botnet, exfiltrating information from the victim, upload commands or configure a remote backdoor [3].

Covert channels can be created by hiding information in different portions of the traffic. Specifically, an attacker can directly inject data in packets composing the legitimate stream, e.g., by storing secrets in unused bits of the header or by further compressing the payload to free space. Secrets can be also cloaked via suitable encoding schemes able to alter some properties of the traffic, e.g., via modulating the inter packet time [4]. The research community has already investigated covert channels targeting IPv4, but it has largely neglected the popularity gained by IPv6 in last years [4–6]. In fact, IPv6 offers to attackers various options for hiding data [7], either by using transitional mechanisms or by exploiting peculiarities of the traffic in real-world deployments [8].

The detection of a network covert channel is a nontrivial and poorly generalizable problem [4]. Spotting hidden communications within a bulk of network flows typically requires to implement attack-specific methodologies or to perform deep packet inspection, which poses scalability problems [9]. Moreover, security tools can not detect IPv6 covert channels out of the box [8], and many of them even have issues in handling IPv6 traffic as well as conversations exploiting v4/v6 transitional mechanisms [10].

The detection of covert channels targeting IPv6 is of prime importance today to fully assess security of modern network scenarios and to mitigate the advancement of stegomalware and other information hiding attacks [1–3]. Nevertheless, scalability of the approach should be considered as a design constraint, since inspection processes should not penalize legitimate traffic flows, e.g., by adding additional delays or disrupt the perceived Quality of Experience [1, 4, 9]. To this aim, we introduce `bccstego`, a framework that can be used for detecting network covert channels in the header of IPv6 packets. Specifically, `bccstego` allows to create usage statistics for specific fields, which can reveal anomalous patterns. The

ARES '21, August 17–05, 2021,

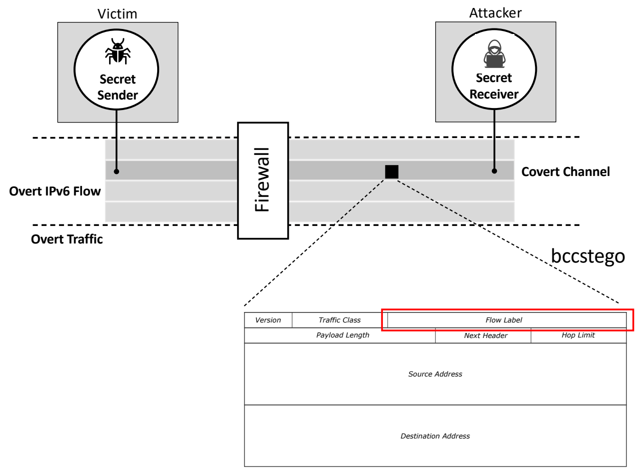


Figure 1: Reference scenario and attack model for the use of a network covert channel.

tool exploits the BPF Compiler Collection (BCC) framework, which facilitates the creation and injection of extended Berkley Packet Filter (eBPF) programs within the Linux kernel, as well as the collection of measurements in user-space. This turns into a very flexible approach, which can be easily extended to other protocols, such as IPv4, TCP and UDP. Moreover, our method does not introduce instability in the kernel and, compared with other solutions (e.g., flow monitoring), its lightweight nature allows to scale well with a growing number of flows. In fact, `bccstego` has a constant memory consumption and a processing overhead of few CPU instructions per packet.

The contribution of this work is twofold: *i*) a tool that collects statistical indicators for IPv6 traffic, which can be used for spotting network covert channels and, *ii*) a performance evaluation that takes into account the impact in terms of packet processing overhead and CPU/network footprint.

The rest of the paper is organized as follows. Section 2 briefly reviews the problem of network covert channels, with specific focus on those exploiting fields in the IPv6 header. Section 3 explains the methodology to collect measurements, and the reasoning behind this choice. Then the program currently available in `bccstego` is described in Section 4, while functional and performance evaluation is reported in Section 5. A brief overview of related work and alternative technologies is given in Section 6. Finally, conclusions and plans for future work are discussed in Section 7.

2 NETWORK COVERT CHANNELS TARGETING IPV6

As said, a network covert channel allows two peers (commonly defined as secret sender and secret receiver) to covertly communicate through the Internet (see Fig. 1). To this aim, the secret sender hides data by injecting information in packets of an overt traffic flow. The overt flow, acting as the carrier for the secret information, can be eavesdropped (in this case, the two secret endpoints act in a Man-in-the-Middle manner) or generated artificially. The ultimate goal of a network covert channel is to evade blockages (e.g., a forwarding rule that prevent communications from/to specific range of network addresses) or to elude the detection from security tools (e.g., a firewall). In the general model, we consider an attacker who wants to covertly communicate with a victim host previously infected via a suitable vector, e.g., phishing [1].

Despite the wide array of options, the most popular mechanisms for the creation of an IPv6 covert channels are those targeting its header, as they provide a good tradeoff among robustness (i.e., how the channel can resist against delays, errors and deliberate manipulations from a security tool), capacity (i.e., how much secret information can be sent per time unit) and undetectability (i.e., how the channel is difficult to spot) [4, 6–8]. Specifically, we consider two complementary methodologies in our work, which either fill the entire field or modulate its original value with a secret.

There are two main examples for injecting information by rewriting an entire field [7, 8, 11]. The first exploits the `Traffic Class`, which specifies the service expected from the network. The information contained in this 8 bit long field can be replaced with hidden data to create a covert channel with a bandwidth of 8 bit per packet. The second exploits the `Flow Label`, which helps network nodes to route traffic towards the most appropriate path. The 20 bit long labels are generated in a pseudo-random manner and can be replaced with hidden data, leading to a covert channel with a capacity of 20 bit per packet.

Another popular approach is based on the modulation of the original values of a field. Among the others, the `Hop Limit` has been considered in many works [7, 8, 11]. It is 8 bit long and defines the maximum number of nodes that can be traversed by the packet. A channel of a capacity of 1 bit per packet can be created by increasing or decreasing the value of this field for consecutive packets. The secret is then decoded by comparing the received values [7, 11].

Despite the number of alternative techniques to create covert channels, existing security tools are not able to spot their presence (see, e.g., [9] and the references therein). Therefore, we are developing a framework that could be easily extended to detect as many network channels as possible. An important design requirement was to not disrupt legitimate traffic or penalize IPv6 conversations, by adding further delays or increasing the packet loss [9]. For this reason, `bccstego` defines a framework made of an eBPF program and a user-space utility, which are combined to inspect packets, collect field usage statistics, and report them for analysis. This approach simplifies the integration with standard kernel operation, reduces the overhead, improves the scalability, and facilitates the portability to different Linux systems. In this paper, we describe our first tool, which covers the relevant fields in the standard IPv6 header that were briefly introduced above.

3 COLLECTING STATISTICS ON IPV6 HEADER FIELDS

The most effective and straightforward way to detect covert channels in the IPv6 header requires to track the values assigned to relevant fields within the same network flow, an approach that we can indicate as "flow tracing." Usually, the `<src addr, dst addr, protocol, src port, dst port>` tuple is used for identifying a conversation. For each stream, several parameters are recorded, e.g., the number of packets, number of bytes, average inter-packet

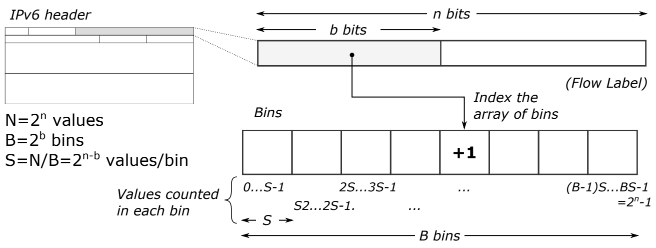


Figure 2: Mapping field values to bins.

delay, and the status of the flags. It would be simple to extended such approach to also consider fields vulnerable to covert communication attempts (namely `Flow Label`, `Traffic Class` and `Hop Limit`) and use this information to detect anomalous usages. However, the overhead due to tracing increases linearly with the number of flows, and may be unfeasible for large Internet links. Indeed, it is well-known that common tools for network monitoring cannot sustain high bitrates, and sometimes make use of sampling techniques for "estimating" the active flows [12].

Based on this consideration, we introduce an alternative technique able to scale independently of the number of flows. Rather than keeping the "state" for each flow, we only measure general statistics about the usage of vulnerable fields. More in details, we count the number of occurrences for the different values that a given field assumes. To make this approach scalable, multiple values may be grouped together into what we call a "bin", and a single counter is used for the whole group. We will refer to this technique as "counters" method to emphasize the different approach with flow tracing previously outlined.

Fig. 2 depicts our approach. Specifically, we consider $N = 2^n$ possible values for a given field, where n is its length in bit, hence $n = 20$ for `Flow Label`, and $n = 8$ for `Traffic Class` and `Hop Limit`. Such values are split into $B = 2^b$ bins, thus there are $S = 2^{n-b}$ values that are grouped into the same bin. In our design, the number of bins is always a power of 2, which is necessary to have uniform bins. In addition, with this constraint, mapping the value of a field to the corresponding bin reduces to a simple bitwise operation, i.e., a prefix matching where the first b bits of a field value are used to index the corresponding bin. Let us consider a numerical example, with $b = 8$ and the current packet bearing the value of `0xA59B8` in the `Flow Label` field. Since $b = 8$, the resulting 256 bins are indexed into an array ranging from 0 to 255 and each bin counts the occurrences of 2^{12} different `Flow Label` values. To find the bin to increment for the given `Flow Label` value, we use its first 8 bits, i.e., `0xA5`, as the index for the array, i.e., the counter to be incremented is the one with the index equal to 165.

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

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

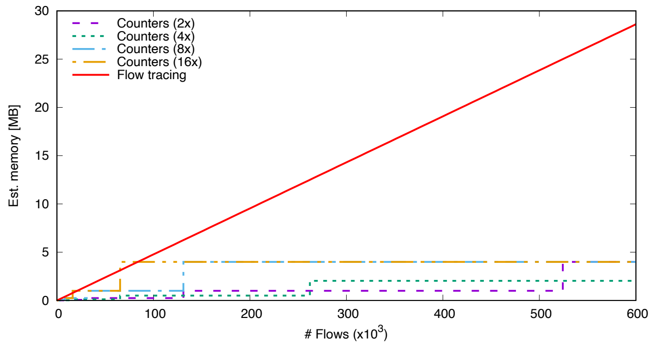


Figure 3: Estimation of memory consumption with different number of flows.

When using counters, only 32 bits of memory are required for each bin, independently of the length of the considered field. However, the number of bins B is not fixed, and we have to select this value in an appropriate way to make the comparison fair. Indeed, for the `Flow Label`, using a number of bins equals to the entire value space (e.g., $b = 20$) would led to excessive overhead. From our experiments we found the empirical rule of thumb that the number of bins B should be at least twice the average number of flows, in order to capture meaningful trends that could be used for the detection of anomalies caused by a network covert channel nested within a flow. To be more conservative in our estimation, we considered different scenarios, where the number of bins was 2, 4, 8, and 16 times the number of active flows. For the `Traffic Class` and the `Hop Limit` the memory consumption accounts to 8192 bits in the worst-case scenario, namely when 256 bins are used (one bin for each value).

Fig. 3 compares the estimated memory consumption of our approach versus flow tracing while varying the number of active flows. As shown, the memory requirement for standard flow tracing increases linearly with the number of flows (i.e., each flow requires a 400 bit long record), whereas our method scales much better and the estimated memory consumption is larger only in case of few flows.

4 THE BCCSTEGO FRAMEWORK

Our main objective is to design a tool able to run with a low execution footprint the inspection in different environments, both physical and virtual. Since the range of possible covert channels is virtually unlimited, extensibility is an important design constraint, mainly for handling additional protocols or considering new attacks. Based on such considerations, we found the Linux-based eBPF to be the best suited technology that fits our requirements. In this section, we will introduce the eBPF framework, the used library, as as well as the implementation details of the first tool of our bccstego framework.

4.1 The eBPF Framework

Originally conceived as an efficient mechanism for packet filtering, the BPF has recently widened its scope, being now able to define small tasks triggered by the reception of a packet or the execution of a kernel function (not limited to system calls). Roughly speaking, an eBPF program consists of two parts: *i*) a *hook* that triggers its execution, and *ii*) a list of instructions to be executed. eBPF programs run in a dedicated virtual machine within the Linux kernel, hence they have their own instruction set. They can be loaded dynamically, but for security and stability reasons, programs have limited access to system resources. Thus, the only way to interact with an eBPF program is through *maps*, which are shared-memory regions. Therefore, the typical development pattern includes both the eBPF program and a user-space utility for loading the program in the kernel as well as pushing/collecting data.

There are different types of eBPF programs, based on the specific context where they are executed. For our purposes, we develop programs for the traffic control subsystem (usually indicated as `tc`), which inspects network packets via the special *clsact* qdisc. This approach gives access to both ingress or egress traffic, and to richer kernel metadata than the eXpress Data Path subsystem.[1]

4.2 The BPF Compiler Collection

Although the implementation of an eBPF program consists of a limited number of instructions, there may be a non-negligible overhead in the external constructs that are necessary to compile the code, load it into the kernel, and exchange data. Among the alternative loaders available (e.g., `bpftool`, `tc` and `ip` utils), BCC[2] emerged as a flexible yet powerful framework for running eBPF programs, including a rich collection of tools for investigating the performance of the operating system.

The programming model for BCC tools revolves around a Python class delivering functionalities for compiling, loading, and running eBPF programs. The class also takes care of creating shared maps, and provides specific methods to read and write data. The user-space portion of the code is therefore written in Python, but BCC also offers some bindings for the Go language. The source code of the eBPF program is usually embedded into the Python script (e.g., it is statically-stored as string within the module), which simplifies the portability of the application. Alternatively, eBPF programs can also be loaded from an external file.

4.3 The ipv6stasts.py tool

The `bccstego` is an umbrella for collecting various tools targeting specific traits of the traffic and provide support for the detection of network covert channels. The idea is to share a common pattern for parsing packets and collecting data, while different eBPF programs are developed for specific protocols or steganographic threats.

Currently, we provide the `ipv6stats.py` tool that builds usage statistics for multiple header fields, as described by the "counters" methods introduced in Sec. 3. The name clearly indicates that the current version specifically targets IPv6 packets, but we plan to enrich the framework for covering a broader set of protocols[3] .

Figure 4 depicts the build process for `ipv6stats.py`, which is explicitly designed to facilitate the maintenance of the complementary programs (i.e., the eBPF filter and user-space utility). There are two distinct files: `bpfprog.c` providing the skeleton of the eBPF program and `userprog.py`, which is the user-space utility written in Python. A `Makefile` helps to automatize the merge of the different components and build the monolithic `ipv6hstats.py` exucatable Python module. As soon as new programs will be added to `bccstego`, the same implementation pattern will be followed.

Since the eBPF code is run for each packet, it is important to use as few instructions as possible, mainly to avoid unnecessary computation overheads and delay. For this reason, the `ipv6hstasts.py` script dynamically creates the eBPF code for monitoring a specific IPv6 header field, which is indicated on the command line as a parameter. The current version is able to create, starting from the common skeleton, programs for gathering data about the `Flow Label`, `Traffic Class` and `Hop Limit` fields. Other parameters that can be given from the command line include the number of bins B, the network interface to be monitored, the direction of the traffic, the sampling interval, and the name of the file for saving the obtained statistics.

5 NUMERICAL RESULTS

To evaluate the performances of the `ipv6hstats.py` tool in the `bccstego` framework, we prepared a testbed composed of three virtual machines running Debian GNU/Linux 10 (kernel 4.20.9), with 1 virtual core and 4 GB of RAM: two machines exchanged traffic, while the third one acted as a router and ran our software. All the virtual machines were running on a 3.60 GHz Intel i9-9900KF host with 32 GB of RAM and Ubuntu 20.4 (Linux kernel 5.8.0). To test our tool in real-world network conditions, the overt IPv6 traffic load was created by replicating (via the `tcpreplay` tool) traffic collected on a OC192 link between Sao Paulo and New York on January 17, 2019 from 14:00 to 15:00 CET, and made available by the Center for Applied Internet Data Analysis[4]. The sampling interval (namely, the frequency with which the user-space program reads the counters updated by the eBPF) was set equal to 1 second.

Though our tool provides the current values for every bin, this information is complex to depict and not intuitive to analyze. Therefore, we use a more compact parameter, namely the number of bins that were incremented since last read. This is shown in Figure 5 and indicated as "number of changed bins." In case of `Flow Label` (see, Figure 5(a)), this parameter provides an approximate information about the volume of active IPv6 flows populating the overt traffic. In fact, every packet belonging to the same IPv6 flow is expected to use the same `Flow Label` value. When an attacker uses this field to

[1]Indeed, XDP allows access to packets before the `struct sk_buff` is allocated. The only context for XDP programs is a couple of pointers that delimit the packet boundaries in memory. For this reason, XDP programs are particularly indicated for DDoS mitigation or load balancing, because this kind of activities can often avoid the expensive overhead of allocating `sk_buff` structures.

[2]BCC, available on line: https://github.com/iovisor/bcc. Last Accessed: March 2021.

[3]The `bccstego` framework is open-source and publicly available on GitHub: `link removed to avoid the disclosure of the identities of the authors.`

[4]The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) - Used traces: Jan. 17th 2019. Available online: https://www.caida.org/data/monitors/passive-equinix-nyc.xml [Last Accessed: April 2021].

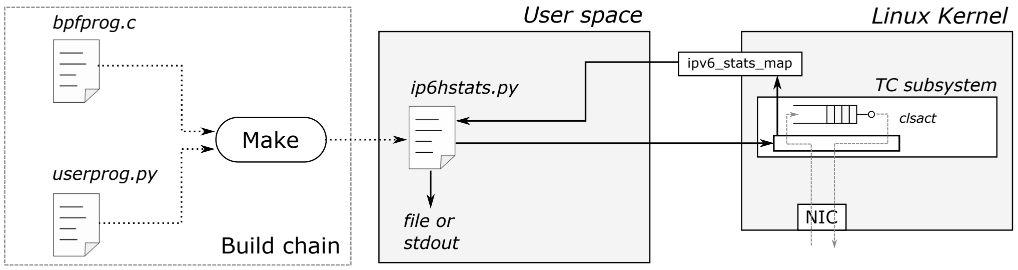
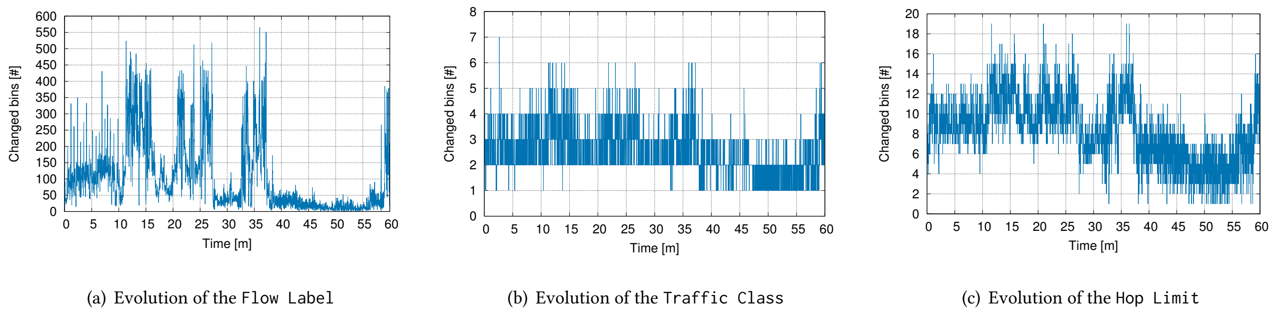
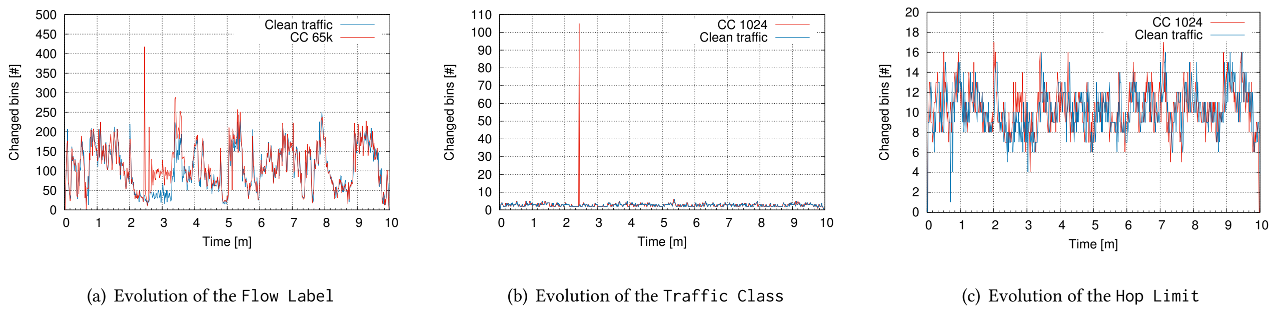


Figure 4: Architecture of the `ipv6hstats.py` tool.



(a) Evolution of the `Flow Label` (b) Evolution of the `Traffic Class` (c) Evolution of the `Hop Limit`

Figure 5: Number of changing bins of the observed traffic when gathering data for different fields.



(a) Evolution of the `Flow Label` (b) Evolution of the `Traffic Class` (c) Evolution of the `Hop Limit`

Figure 6: Different behaviors of the number of bins changed between clean overt traffic and traffic containing a hidden communication for different fields.

bear some secret, the number of changing bins will abnormally increase, hence this parameter may be directly used for the detection of covert channels. A similar consideration can be drawn for the evolutions observed for the `Traffic Class` and `Hop Limit`. Specifically, Figure 5(b) clearly shows that the number of values observed for the `Traffic Class` is limited. Thus, an attacker wanting to inject data in this field without producing visible alterations should adopt some form of encoding (e.g., mapping `1` and `0` values into a sequence of `Traffic Class` values that are present in the overt traffic), or he/she should keep the data rate as small as possible (see [8] for a thorough investigation on the embedding capacity of real-world IPv6 traffic). Similarly, using the `Hop Limit` as a carrier for secrets requires the attacker to modulate values without deviating too much from the observed average value (see Figure 5(c)).

Concerning resources used by the `ipv6hstasts.py` tool, we considered performances of both the eBPF and the user-space programs. The eBPF code is composed of about 120 assembly instructions, which are executed only when a packet is processed. As a consequence, providing a simple estimation for the CPU usage is not straightforward. Instead, we can provide an estimation of the additional latency introduced when processing a packet. It turned out that such a quantity is very small, 104.48 ns, on average. The maximum and minimum execution times observed were 19, 715 ns and 49 ns, respectively. To give an idea of the impact of the latency introduced by our eBPF code on a realistic case, we measured the

Table 1: CPU and memory usage for the user-space program.

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

total delay introduced when transferring a file of 1.2 Gbytes. Results indicate a very minor impact, as the additional delay was equal to ~7 ms. As regards the memory usage, the stack size is limited to 512 bytes. Instead, the amount of shared memory depends on the number of bins, but it is insensitive to the length of the monitored field, as already discussed in Sec. 3. Accordingly, the maximum memory occupancy occurs for 2^{20} bins, and it is equal to ~8 MB, which is anyway rather small. For lower number of bins (i.e., in the range of $4 - 8$ entries), memory consumption is constant and equal to 4 KB. Between these two ends, it increases proportionally to the number of bins.

Coming to the user-space program, we measured both the memory and CPU usage. Table 1 reports the data obtained via the system tools (`top` and `time`). Similar to the case of eBPF code, the size of the shared memory area impacts on resource consumption, since data is copied in user-space. For the sake of brevity, we only consider a limited number of bins, corresponding to what used in Figures 5 and 6. For the case of CPU usage, the relevant parameter is the sampling interval, i.e., the time frame between consecutive reads from the user-space utility, denoted as "interval" in the table. As it can be seen, the larger number of bins, the higher the utilization or resources. For `Traffic Class` and `Hop Limit`, which only uses 256 bins, there is no meaningful difference in CPU utilization when changing the sample time.

5.1 Towards the Detection of Covert Channels

Even if `ipv6hstasts.py` and upcoming tools in the `bccstego` framework can be used as general methodologies for investigating different anomalies in network traffic, our prime goal is to use them for the detection of network covert channels. To evaluate this possibility, we conducted an additional round of tests. Specifically, we used the same testbed and traffic conditions of the previous round, but we added two malicious endpoints that hide a secret in the different fields of the IPv6 header. The covert channel is implemented by intercepting over packets with NetfilterQueue 0.8.1 and re-writing the header fields with an ad-hoc Python scripts which uses Scapy 2.4.3. The two endpoints injected data within an SCP file transfer with a rate of 500 kbit/s.

Initially, we injected the secret in the `Flow Label` of an IPv6 conversation. The length of the secret message was set to 65 kbyte, which is representative of a chunk of sensitive data or the retrieval of additional attack routines. Figure 6(a) shows the number of changed bins, while comparing to the "clean" scenario (namely, when no covert channel is present). For the sake of clarity, the figure is narrowed to a timeframe of 10 minutes. We have already noted how, when no covert channel is present, changes in the number of bins indicate the arrival of new IPv6 conversations. This is due to the "natural" evolution of randomly-generated `Flow Label` values that are expected to fall under different bins during the sample period. This approximation is more precise when the number of new flows is smaller than the number of bins (hence, B and the sampling interval must be chosen accordingly). Based on this consideration, if the number of used bins suddenly increases, this is an indicator that a hidden communication may be on-going.

Different considerations can be done for other fields of the IPv6 header. For the case of a channel in the `Traffic Class`, the limited amount of values used in practice makes the detection trivial, as indicated by the major "spike" at ~3 minutes shown in Figure 6(b). To elude the detection, the attacker should be able to use a suitable encoding or to slow the channel down to only few bits per minute. The "modulating" flavor used to hide a cover channel in the `Hop Limit` makes the detection more difficult. As depicted in Figure 6(c), the altered behavior is less visible since the secret information is not directly injected in the field and the alterations are spread over the various bins in a more regular manner.

6 RELATED WORK

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

In more detail, the most effective approach is kernel bypass, which replaces the networking stack with an alternative path. Notable examples are PF_RING [13] and Netmap [14], which map NIC memory and registers to userland to avoid the need of copying packets. To support such a paradigm, applications must re-implement common networking utilities and protocols. To partially overcome this issue, DPDK [15] provides a large set of libraries for common packet-intensive tasks, whereas OpenOnload [16] uses a hybrid architecture, which dynamically selects between user-space and kernel mode for any network flow. To further reduce the impact of context switches in the hardware, Vector Packet Processing [17] exploits the persistence of common information in the processor caches. In this case, it collects and processes large batches of packets (called vectors). From the viewpoint of supporting security appliances and operations, [13–17] have been largely used in middleboxes for intrusion detection, firewalling, flow monitoring, and mitigation of denial of service attacks.

Even if kernel bypass is a very effective mechanism for simple networking processes (e.g., packet forwarding and routing), the implementation of generic communication channels is not trivial. Hence, many frameworks make use of kernel bypass technology to create common processing patterns: Click [18], BESS [19], Snabb [20], just to mention a few. In this case, the adoption of fixed processing patterns may jeopardize the implementation of tailored monitoring and detection features.

More recently, eBPF was introduced to go beyond simple packet filtering. Even if it cannot reach the performance of kernel bypass mechanisms, it represents a very flexible and efficient solution for making custom operations on the traffic processed by the host. eBPF has been mainly conceived for investigating the kernel performance, while security-related tools are largely missing. Interestingly, many eBPF-based tools are being integrated in the Cilium platform [21]. The flexibility of eBPF and the possibility to precisely monitor and trace the kernel make this framework a really promising technology for discovering and investigating a large variety of stegomalware [22, 23] within single hosts or complex digital infrastructures [24].

7 CONCLUSION AND FUTURE WORK

In this paper we have introduced our framework `bccstego`, and the first tool we developed for collecting statistics on IPv6 header fields. Differently from other approaches, we leveraged in-kernel code augmentation to reduce the development effort without impacting the packet processing performances provided by Linux. Scope of the tool is to provide a foundation to support the detection of stegomalware leveraging network covert channels. In this perspective, `bccstego` should not be conceived as a tool working "out of the box" but it has to be considered as the part of a larger framework that aggregates information from complementary sources.

Future work will aim at extending the tools to other protocols, as well as to test the performances of `bccstego` when jointly used for feeding algorithms aiming at revealing the presence of network covert channels.

ACKNOWLEDGMENT

This work has been supported by the EU Project ASTRID, Grant Agreement No 786922, the EU Project GUARD, Grant Agreement No 833456, and the EU Project SIMARGL, Grant Agreement No 833042.

REFERENCES

[1] L. Caviglione, M. Choraś, I. Corona, A. Janicki, W. Mazurczyk, M. Pawlicki, and K. Wasielewska, "Tight arms race: Overview of current malware threats and trends in their detection," *IEEE Access*, vol. 9, pp. 5371–5396, 2021.

[2] W. Mazurczyk and L. Caviglione, "Information hiding as a challenge for malware detection," *IEEE Security & Privacy*, vol. 13, no. 2, pp. 89–93, 2015.

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

[4] S. Zander, G. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 44–57, 2007.

[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] W. Mazurczyk and L. Caviglione, "Steganography in modern smartphones and mitigation techniques," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 334–357, 2014.

[7] N. Lucena, G. Lewandowski, and S. Chapin, "Covert channels in ipv6," in *Int. Workshop on Privacy Enhancing Technologies*. Springer, 2005, pp. 147–166.

[8] W. Mazurczyk, K. Powójski, and L. Caviglione, "IPv6 covert channels in the wild," in *Proceedings of the 3rd Central European Cybersecurity Conference*, 2019, pp. 1–6.

[9] L. Caviglione, "Trends and challenges in network covert channels countermeasures," *Applied Sciences*, vol. 11, no. 4, 2021.

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

[11] G. Lewandowski, N. Lucena, and S. Chapin, "Analyzing network-aware active wardens in ipv6," in *Int. Workshop on Information Hiding*. Springer, 2006, pp. 58–77.

[12] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 245–256, 2004.

[13] L. Deri, "Improving passive packet capture: Beyond device polling," in *Proceedings of SANE 2004*, Amsterdam, The Netherlands, Sep., 27th – Oct., 1st, 2004, pp. 85–93.

[14] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, Jun.12th–15th, 2012, pp. 101–112.

[15] "DPDK programmer's guide," Technical Documentation, February 2021. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/

[16] S. Pope and D. Riddoch, "Introduction to openonload—building application transparency and protocol conformance into application acceleration middleware," Whitepaper, 2011. [Online]. Available: http://www.moderntech.com.hk/sites/default/files/whitepaper/SF-105918-CD-1_Introduction_to_OpenOnload_White_Paper.pdf

[17] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-speed software data plane via vectorized packet processing," *IEEE M COM*, vol. 56, no. 12, pp. 97–103, December 2018.

[18] E. W. Kohler, R. T. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek, "The click modular router," *Publication: ACM Transactions on Computer Systems*, vol. 18, no. 3, August 2000.

[19] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[20] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, "Snabbswitch user space virtual switch benchmark and performance optimization for nfv," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, San Francisco, CA – USA, Nov., 18th–21st, 2015, pp. 86–92.

[21] L. Makowski and P. Grosso, "Evaluation of virtualization and traffic filtering methods for container networks," *Future Generation Computer Systems*, vol. 93, pp. 345–357, April 2019.

[22] A. Carrega, L. Caviglione, M. Repetto, and M. Zuppelli, "Programmable data gathering for detecting stegomalware," in *6th IEEE Conference on Network Softwarization (NetSoft)*, Ghent, Belgium, Jun., 29th – Jul., 3rd, 2020.

[23] 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, p. 108010, 2021.

[24] 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.