pcapStego: A Tool for Generating Traffic Traces for Experimenting with Network Covert Channels

The increasing diffusion of malware endowed with steganographic and cloaking capabilities requires tools and techniques for conducting research activities, testing real deployments and elaborating mitigation mechanisms. To investigate attacks targeting network and appliances, a core requirement concerns the availability of suitable traffic traces, which can be used to derive mathematical models for simulation or to develop machine-learning-based countermeasures. Unfortunately, the young nature of threats injecting secrets or cloaking their presence within network traffic, the high protocol-dependent nature of the various embedding processes, and privacy issues, prevent the vast diffusion of datasets to perform research. Therefore, in this paper we present `pcapStego`, a tool for creating network covert channels within `.pcap` files. This approach has two major advantages: it allows to prepare large datasets starting from real network traces, and it generates "replayable" conversations useful for both emulating attacks or conduct pentesting campaigns. To prove the effectiveness of the tool, we showcase the generation of network covert channels targeting IPv6 traffic, which is gaining momentum and it is expected to be a major target for future attacks.

CCS Concepts: • **Networks** → *Network experimentation*; Network reliability.

Additional Key Words and Phrases: covert channels, IPv6, dataset generation, detection, traffic traces

ACM Reference Format:

. 2021. pcapStego: A Tool for Generating Traffic Traces for Experimenting with Network Covert Channels. In *3rd International Workshop on Information Security Methodology and Replication Studies, August 17–20, 2021, All-digital.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Information hiding and steganographic techniques are increasingly adopted by attackers to develop a new-wave of malware able to remain unnoticed or evade security mechanisms [16]. Specifically, they can be used to conceal malicious payloads or attack routines in innocent-looking images, deploy stealthy multi-stage loading architectures or obfuscate code to resist forensics investigations [1]. Despite the wide range of possibilities, malware primarily deploys information hiding to implement covert channels [1, 16]. According to the definition introduced by Lampson in 1973, covert channels are *"[channels] not intended for information transfer at all"* [12]. For instance, a covert channel can be created by manipulating a shared resource (e.g., a lock or the space available in the local file system) to let two processes to communicate even if confined within different sandboxes [11]. However, a major application concerns the creation of hidden communication paths to allow endpoints secretly exchanging data through the Internet [16, 22]. In this case, covert channels are used to exfiltrate information from the victim towards a remote server controlled by the attacker, implement command & control functionalities, as well as to configure backdoors or orchestrate a botnet without being detected [1, 5, 16].

Manuscript submitted to ACM

Despite the increasing volume of attacks, the magnitude of economical losses, the degree of sophistication, and the growing attention from security-oriented software firms[1], threats leveraging network covert channels are often neglected for a threefold reason. First, the emerging nature of information-hiding-capable malware still requires precise investigation methodologies and conceptual devices. Second, the creation of a covert channel is tightly coupled with the specific protocol/feature exploited to conceal the communication. Third, network data usually contains confidential information requiring to adhere to strict gathering and collection rules to not disrupt privacy of users. As a consequence, the number of malware samples isolated in production-quality deployments is limited and datasets containing network traffic or execution traces do not consider information-hiding-capable threats. Even if efforts from the research community for sharing data are multiplying (see, e.g., [18] and the references therein), the majority of publicly available collections appear to be tailored for developing intrusion detection systems able to counteract "classical" attacks like Denial of Service, flooding, port scanning, and botnet orchestration. Moreover, public datasets often require a non-negligible preprocessing effort to isolate or label information of interest [21]. Typical workarounds for the lack of traces capturing attacks leveraging network covert channels exploit artificially-generated traffic, non-weaponized malware, and simulations. Despite the level of accuracy, real-world data is almost mandatory to prove the correctness of laboratory trials or to evaluate the performance of a mitigation technique [10]. Moreover, the adoption of AI to address security issues (e.g., detection, classification or reverse engineering of malicious code) demands for relevant volumes of data, which should be organized in a standardized form for reproducing and comparing the different techniques [9]. Therefore, in this paper we introduce `pcapStego`, which is a tool for creating network covert channels in `.pcap` traces. Compared to synthetic traffic generation or crafting toy attacks, `pcapStego` offers the following benefits: *i*) data is injected within real traffic to consider production-quality scenarios or perform "what if" analyses; *ii*) the creation of covert channels happens offline enabling to prepare large datasets or exchange attack templates for reproducing experiments; *iii*) `.pcap` traces can be replayed for pentesting purposes or to route traffic through real devices or laboratory equipments.

Despite the lack of datasets, literature already proposes several works investigating network covert channels targeting IPv4 supplied with prototypal implementations [15, 22]. Instead, the IPv6 counterpart has been only investigated in a theoretical manner [13]. For this reason, this first release of `pcapStego` aims at filling such a gap and implements functionalities to inject data within IPv6 traffic, which is expected to become a prime vector for hiding malicious conversations owing to its momentum and rich set of functionalities [1, 5, 16]. Specifically, the tool allows to create network covert channels targeting the `Flow Label`, `Traffic Class`, and `Hop Limit`, which are the most effective containers for secret data when considering real network deployments [17].

Summing up, the contributions of this paper are: the definition of a methodology for investigating covert channels within a realistic traffic/network ecosystem and the demonstration of the effectiveness of injecting information in preexistent `.pcap` data.

The remainder of the paper is structured as follows. Section 2 provides a background on network covert channels, with emphasis on those targeting IPv6 as well as the considered security problem. Section 3 discusses the architectural blueprint and design choices for the proposed tool, while Section 4 showcases results obtained in field trials. Lastly, Section 5 concludes the paper and portraits future research directions.

[1] See, [REMOVED FOR BLIND REVIEW] for an updated list of attacks and malware using steganography and information hiding techniques.

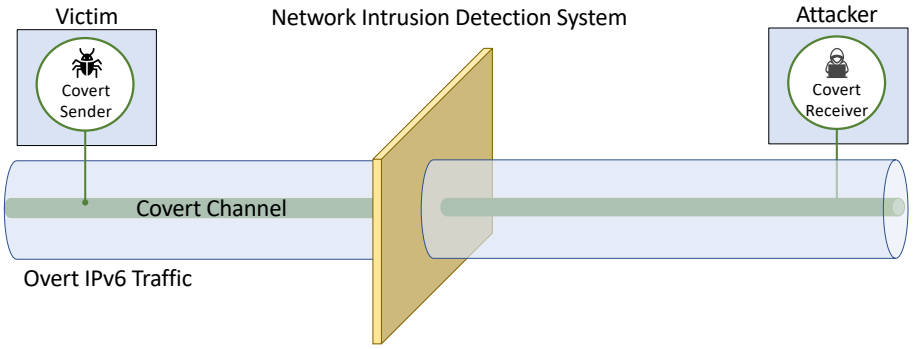

Fig. 1. Reference scenario and attack model for the use of network covert channels.

2 COVERT CHANNELS AND ATTACK MODEL

The use of covert channels to cause a variety of security hazards has been known for decades, mainly to leak data between CPUs, to map the underlying hardware or to discover the presence of virtualization layers or hypervisors [2]. The exploitation of network covert channels has become popular in recent years, especially since a traffic flow is almost boundless and it is not uncommon to have network infrastructures with flows that last several hours and can hide information to support advanced persistent threats [5, 16]. In general, a network covert channel is created between two covert endpoints (defined as *covert sender* and *covert receiver*, respectively) wanting to remotely communicate in a hidden manner. To this end, a legitimate, overt network flow is used as the carrier to contain the secret information. Figure 1 shows the reference scenario and the considered attack model. Specifically, it depicts an attacker wanting to remain unnoticed or bypass a network intrusion detection system for communicating with the host of the victim, e.g., to activate a backdoor or establish a network path fo exfiltrating sensitive data.

Covert channels within a network flow can be created in two manners. In *timing channels* the sender encodes information by modulating a trait of the traffic, for instance it alters the delay experienced by two consecutive packets composing the flow. Instead, in *storage channels* the secret information is directly embedded in network packets without disrupting the legitimate conversation. As a possible example, data can be hidden in unused header fields. Currently, `pcapStego` can create a hidden communication within an IPv6 conversation by directly injecting data in two fields of the header (i.e., the `Traffic Class` and the `Flow Label`) or by manipulating the behavior of the `Hop Limit`. Specifically, the tool allows to create network covert channels in the following parts of the IPv6 header:

- `Traffic Class`: this 8 bit long field is used to specify the type of service expected from the network. In real-world deployments, it is seldom used, thus it can be manipulated to contain arbitrary information [13, 17].
- `Flow Label`: this 20 bit long field supports the network in routing duties. In general, it can be altered without being noticed by firewalls or other security tools [17].
- `Hop Limit`: this 8 bit long field specifies the maximum number of intermediate nodes that a datagram can traverse. A secret can be encoded by modulating its value for consecutive packets, i.e., bit `1` and `0` are encoded by increasing or decreasing the value of the `Hop Limit` by a given threshold value [13].

Regardless the selected embedding mechanism, an attacker usually seeks for a covert channel robust, hard to detect, and with the highest possible bandwidth. Luckily, such properties cannot be maximized at once, since they are coupled via a magic triangle rule, i.e., it is not feasible to increase a performance index without lowering the remaining two [8].

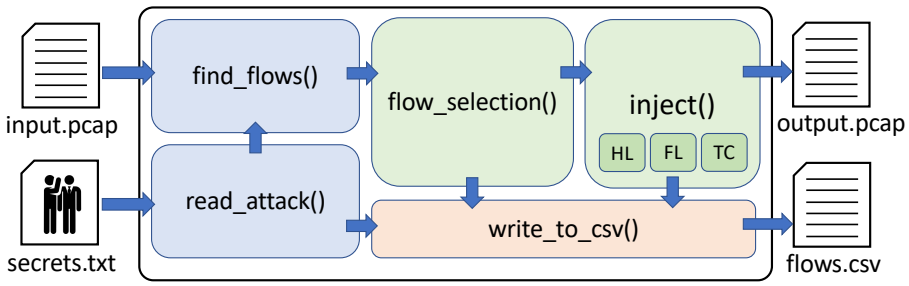

Fig. 2. Software architecture of the pcapStego tool.

The steganographic cost is an additional metric allowing to evaluate the degradation experienced by the carrier due to the presence of the covert communication. As an example, for a network covert channel nested in a VoIP conversation, the steganographic cost can represent the alteration of audio quality (e.g., additional delays, reduced SNR performances of the codec, or audible artifacts)[14]. We point out that, the quantification of such metrics by means of simulations is typically a hard task: therefore, being able to conduct tests directly on real hardware/software components could be beneficial when addressing complex and large-scale scenarios (see, [7] for a general discussion on the main hazards to be solved for accurate modelling of complex network infrastructures).

3 THE PCAPSTEGO DISTRIBUTION

The pcapStego suite[2] is organized in two main sets of tools serving different purposes. The first set implements an *interactive mode* allowing the user to manually select flows, the hiding mechanism (i.e., the part of the IPv6 header that has to be targeted for injecting data) and the secret to be transmitted via the resulting channel. In this manner, users can actively experiment with the tool and gain a deeper understanding of the internals at the basis of network covert channels. The second set, instead, implements a *bulk mode*, which can be used to automatize the embedding of data and thus the creation of various covert communications in the target traffic capture. This working mode is especially suited for creating large `.pcap` files containing various channels at once and in an automatized manner, for instance to replicate experiments.

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

3.1 Software Architecture

To implement pcapStego, we used Python 3 and the Scapy 2.4.4 library[3] for manipulating protocol data units, i.e., to alter fields in the IPv6 header for containing the secret data. To process `.pcap` files, our tool relies upon functionalities provided by the `tshark` 3.4.5 network analyzer[4]. Details are hidden to the user since the needed functionalities have been properly encapsulated within the software architecture, which is depicted in Figure 2. As shown, the tool is

[2] [LINK TO SOFTWARE REMOVED FOR BLIND REVIEW]
[3] https://scapy.net
[4] https://www.wireshark.org/docs/man-pages/tshark.html

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

- `read_attack()`: it parses data contained in the `secrets.txt` file. Each entry in the file specifies the information that has to be sent through the covert channel along with the desired injection mechanism. To this aim, the function splits the secret into chunks fitting the size of the selected carrier (e.g., 8 bits for the case of the `Traffic Class`). When running the tool in interactive mode, the function can also parse the secret information to be sent via the covert channel directly from the command line interface.
- `find_flows()`: it parses data from the source `.pcap` file and prepares a "map" of the available IPv6 conversations. As a result, the function provides a snapshot with the conversations having enough steganographic capacity, i.e., it reports flows with a number of packets able to contain the secret, given the specific injection mechanism.
- `flow_selection()`: it allows to identify in a unique manner the traffic flow selected by the user when the tool is running in interactive mode. The function relies upon a 5-tuple composed of `Source IPv6 Address`, `Destination IPv6 Address`, `Source TCP/UDP Port`, `Destination TCP/UDP Port`, and the used `Protocol`. We point out that, such a 5-tuple will be also used to provide the user appropriate information to prepare filters and to track network covert channels when traffic is replayed through the network.
- `inject()`: it performs the injection within the desired traffic feature. In particular, it searches for the packets belonging to the selected flow and changes them according to the injection mechanism, leaving unaltered the remaining features, such as the payload, the addresses and other parts of the header. The `inject()` function is modular and can be extended to create other covert channels. In this case, it is sufficient to add a "case" to an `if-then` structure and prepare a packet with proper invocations of the Scapy library. The function is also responsible for populating the output `.pcap` file, both with unaltered conversations and those containing the covert channel(s).
- `write_to_csv()`: it logs information about the flows containing the covert channels, such as the 5-tuple to identify the conversation, the injection mechanisms and the length of the hidden data. Information are stored in a CSV file (named `flows.csv`), which can be used to automatize the extraction phase, replicate experiments or generate filtering/forwarding rules for field trials.

We point out that, the aforementioned functions are general enough to handle the different working modes available for `pcapStego` (i.e., interactive or bulk). Minimal differences in the architecture of the tool are only due to the need of presenting to the user a suitable interface for selecting IPv6 flows or to identify IPv6 conversations to target when the tool is used in bulk mode. Specifically, when running in bulk mode, the `find_flows()` function automatically retrieves the 5-tuple for all the traffic flows with enough capacity for the creation of the covert channels provided by the user. A thorough discussion on such implementation details has been omitted for the sake of brevity.

3.2 Usage

As discussed, the `pcapStego` provides a command line interface for specifying the behavior of the tool via various flags. When running in interactive mode, the general usage is:

```
python3 injector.py [-r PCAP] [-f FIELD] [-a ATTACK]
```

with the following mandatory parameters: `-r` defines the source `.pcap` file containing the original traffic trace; `-f` specifies the field of the IPv6 header to use to create the covert channel. In this version, possible values are `FL`, `TC` and `HL`, which identify the `Flow Label`, `Traffic Class`, and `Hop Limit`, respectively; `-a` specifies the secret to be sent via the covert channel. The secret can be a string, the name of a file containing data in textual form, or an image. The tool will handle such different cases in an automated manner. We point out that, without loss of generality, in the `pcapStego` software distribution we often refer to the secret information to be sent via the covert channel as the "attack" in order to emphasize that we aim at providing a tool for testing the stealthy delivery of malicious payloads, fileless malware and other threats.

Figure 3 provides an example output for `pcapStego` when used in interactive mode. As shown, the user can directly interact with the tool for choosing the preferred IPv6 flow. The tool also provides a filter for the `Wireshark` network analyzer[5], which can be used to easily identify the covert channel among the bulk of traffic within the `.pcap`.

Lastly, if the tool is used in bulk mode, the `-a` parameter is used for pointing at a `.txt` file containing the attacks to be injected in as many IPv6 conversations (one per `\n`-terminated line).

The `pcapStego` tool also allows to extract the content of a covert communication from a `.pcap` file. This can be used to parse traffic traces replayed through a network intrusion detection system and captured to understand whether the hidden information has been blocked or partially disrupted. To this aim, when running in interactive mode, the general usage is similar to the injection instance:

```
python3 extractor.py [-r PCAP] [-f FIELD] [-p PACKETS]
```

with `-p` used to specify the number of packets to be extracted. Such a parameter is mandatory when in the presence of covert channels implemented via an information hiding scheme unable to "mark" the start and the end of the hidden transmission. When performing extraction in interactive mode, two optional parameters can be further used. The first is `-b`, which defines the number of bits to be considered for the secret. This can be useful when in the presence of an injection mechanism unable to directly map the single character composing the secret (8 bit ASCII, in this version of the tool) in a field of the IPv6 header. For instance, when embedding secrets in the `Flow Label` field, the modulo of the injection is 20 bit, whereas for the `Traffic Class` is 8 bit. We point out that, the injection process is protocol agnostic, thus the user can develop additional signaling information to be embedded along with the secret data to endow the channel with advanced features such as message delimiting and error recovery. Even if the overall injection process of `pcapStego` could be improved, this is not a limitation, since almost the totality of IPv6 channels do not implement features for identifying the start/end of a hidden transmission [13, 17]. The second optional parameter is `-i` and specifies that the secret within the traffic is an image.

For the case of extracting data from a `.pcap` file in bulk mode, the command line interface is simpler and parameters are limited to: `-r` specifies the name of the traffic trace containing the covert channels, and `-i` specifies the name of the `.csv` file with all the information useful to identify the modified flows (i.e., the 5-tuple uniquely pointing at a flow, the length of the secret both in bits and packets, and the used embedding strategy).

[5] https://www.wireshark.org

```
➜ python3 injector.py -r pcap_example.pcap -f FL -a hello_world.txt
Reading the attack...
Number of packets needed: 5
Number of bits needed: 88
Creating tmp files...
Deleting tmp files...
-------------------------
CONVERSATIONS FOUND
     ipv6.flow              ipv6.src              ipv6.dst tcp.srcport tcp.dstport udp.srcport udp.dstport  ipv6.nxt  #pkts
6   0x00000000  d0e7:fb50:fb38:ba5a::  d31c:6ecf:61a7:c2ff::       40085         444           -           -         6      7
16  0x00000000  d31c:d77f:c002:f6f4::   d0e2:1f7b:fcc0:9fb::       39318         443           -           -         6     16
91  0x000f6ea1  d9e1:ec18:5ddc:73ff::  d0e2:3ac0:8083:7c81::         443       37495           -           -         6      5
-------------------------
Choose the flow by its index (leave it blank for the first flow or 'r' for a random choice): 91
-------------------------
Wireshark filter: ipv6.src == d9e1:ec18:5ddc:73ff:: and ipv6.dst == d0e2:3ac0:8083:7c81:: and ipv6.flow == 1011361 and tcp.srcport == 443 and tcp.dstport == 37495
Reading input pcap. This might takes few minutes...
Injecting...
Injection succesfully finished!
```

Fig. 3. Example output provided by the pcapStego tool when used in interactive mode.

3.3 Companion Data

To improve its usability, the pcapStego tool is bundled with some companion files. Specifically, it contains an example traffic trace (i.e., pcap_example.pcap) that can be used to experiment with the tool "out of the box". The trace has been prepared from realistic traffic provided by the Center for Applied Internet Data Analysis (CAIDA)[6]. Even if network covert channels can be also used to enforce the privacy of individuals, to bypass blockages imposed by oppressive regimes, or to protect sources in informative journalism [19], they should be primarily considered tools to implement attacks and empower malware [1, 16]. To this aim, pcapStego also contains a minimal database of attacks/payloads that can be injected in the traffic for simulating the distribution through hidden network communications of some file-less malware and malicious payloads. In this first release of the tool, such a selection of attacks has been created starting from the "Fileless Command Lines" collection[7].

4 EXPERIMENTAL RESULTS

To prove the effectiveness of pcapStego for the preparation of realistic traffic traces containing network covert channels, especially to support the creation of testbeds, repeatable experiments, and suitable datasets to be used with AI-based frameworks, we conduct different field trials.

4.1 Data Injection and Replaying

In the first round of tests, we evaluated the correctness of the embedding/extraction process, including the preservation of the semantic/structural properties of the .pcap file. To this aim, we used realistic IPv6 traffic traces provided by CAIDA[8]. To avoid burdening our trials, we performed a lightweight pre-processing of traffic. Specifically, we removed single-packet flows and ICMPv6 traffic. Concerning the information transmitted via the various covert channels, we considered a command[9] used by the Astaroth Malware, borrowed from the "Fileless Command Lines" collection. Such a command causes the utility WMIC to download a legitimate file containing an obfuscated JavaScript file, which is then executed and launch a malicious Astaroth routine.

[6]The traffic bundled with pcapStego is based on the data collected by CAIDA on a OC192 link between Sao Paulo and New York on November, 2018 from 14:00 to 15:00 CET. The original dump is part of the CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) available online: https://www.caida.org/data/passive/passive_dataset.xml [Last Accessed: May 2021].
[7]Fileless malware implemented via malicious command lines: https://github.com/chenerlich/FCL. [Last Accessed, May 2021].
[8]Traffic dumps used to test pcapStego have been taken from the CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) available online: https://www.caida.org/data/passive/passive_dataset.xml [Last Accessed: May 2021].
[9]`WMIC.exe os get ved5hit39, 25hit8, numberofusers /format:"https://storage.googleapis.com/ultramaker/09/v.txt"`

```
→ tshark -nr original.pcap -T fields -e ipv6.src -e ipv6.dst -e ipv6.flow -e tcp.srcport -e tcp.dstport
   -e ipv6.tclass -Y "ipv6.src == 215e:b70e:104f:e0f4:: and ipv6.dst == 2865:8e9:af2b:cb17:: and
   ipv6.flow == 74195 and tcp.srcport == 80 and tcp.dstport == 11156" | sed -n 10,15p
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000000
```

(a) Input traffic dump.

```
→ tshark -nr stego.pcap -T fields -e ipv6.src -e ipv6.dst -e ipv6.flow -e tcp.srcport -e tcp.dstport
   -e ipv6.tclass -Y "ipv6.src == 215e:b70e:104f:e0f4:: and ipv6.dst == 2865:8e9:af2b:cb17:: and
   ipv6.flow == 74195 and tcp.srcport == 80 and tcp.dstport == 11156" | sed -n 10,15p
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x0000006f
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000073
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000020
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000067
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000065
215e:b70e:104f:e0f4::   2865:8e9:af2b:cb17::    0x000121d3      80      11156   0x00000074
```

(b) Outuput traffic dump.

Fig. 4. Traffic dumps (in `.pcap` format) before and after using pcapStego to implement a covert channel targeting the `Traffic Class` for transmitting a command used by the Astaroth malware.

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

Figure 4 depicts partial dumps for the `.pcap` traces before and after the creation of a covert channel targeting the `Traffic Class` field. For the sake of clarity, the figure only reports six packets of the original and replayed `.pcap` files. As shown in Figure 4(a), before the creation of the channel, the original IPv6 flow is characterized by a `Traffic Class` equal to `0`. Instead, after the creation of the covert channel used to deliver the malicious Astaroth command, the `Traffic Class` changes accordingly. Figure 4(b) reports the values of the `Traffic Class` of packets containing the "`os get`" portion of the entire command, e.g., `0x6f` and `0x73` correspond to `o` and `s`, respectively.

4.2 Generation of AI-friendly Metrics

As hinted, to face the increasing complexity of threats as well as the large-scale and heterogeneous nature of modern network deployments, a common approach is to use some form of AI to reveal malicious communications, identify malware samples, as well as to perform sanitization of traffic and multimedia contents [5]. Moreover, modern software-defined networks offer a large palette of opportunities both in terms of architectural blueprints (e.g., the possibility of developing controllers to be deployed in edge nodes) and of technologies for gathering data [20]. Hence, being able to produce suitable datasets for training and develop machine-learning based frameworks is of prime importance [4, 5, 20].

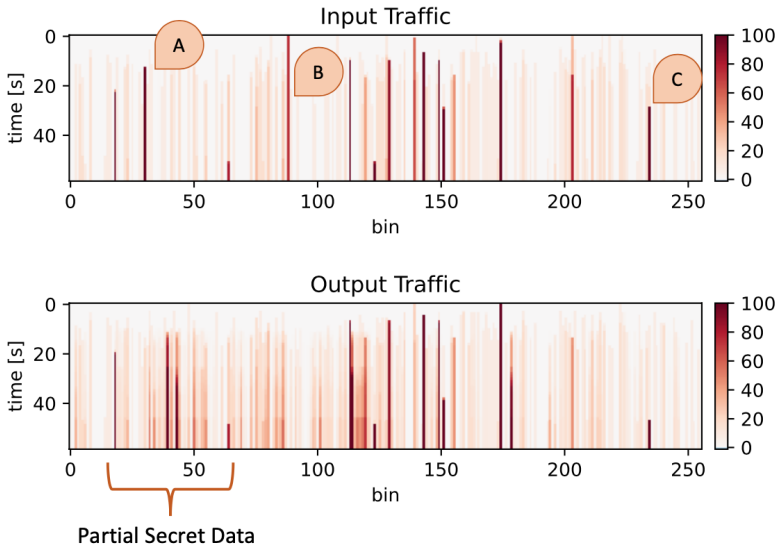

Fig. 5. Heatmaps generated before and after using pcapStego for creating covert channels in the Flow Label.

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

As regards the metric, we bear with heatmaps computed from the distributions of values of the fields targeted by the considered channels. For the Hop Limit, the map has been computed by considering the number of time a specific value has been observed in the overall traffic volume. Instead, for the Flow Label, we adopted a bin-based approach [3]. Specifically, the 2^{20} bit space of the Flow Label has been mapped to a smaller one composed of 2^8 bins (i.e., ranges of values), each one grouping 2^{12} possible values. Such a mechanism prevents scalability issues and "noisy" heatmaps.

To conduct experiments, we prepared a network testbed composed of three hosts. Two hosts exchanged a trace containing the covert channels, which has been "rebuilt" with tcprewrite and then replayed with tcpreplay (denoted as "output traffic" in the figures). A third host is responsible of routing the traffic, collecting the values of the Hop Limit and Flow Label fields and store them on the file-system for further processing. Such a framework leverages the extended Berkley Packet Filter and measurements have been performed with a granularity of 1 second (see, e.g., [6] for a discussion on the use of code augmentation for gather network data to counteract covert channels). To have a reference scenario, we also computed heatmaps of the original .pcap trace, denoted as "input traffic" in the figures.

Figure 5 depicts the heatmaps both for the input and the output traffic. Since each IPv6 conversation is supposed to be identified via its unique Flow Label value, in the following, we refer both to Flow Label and flows interchangeably, except when doubts arise. As shown, pcapStego is used to embed data within flows denoted as A, B and C of the input

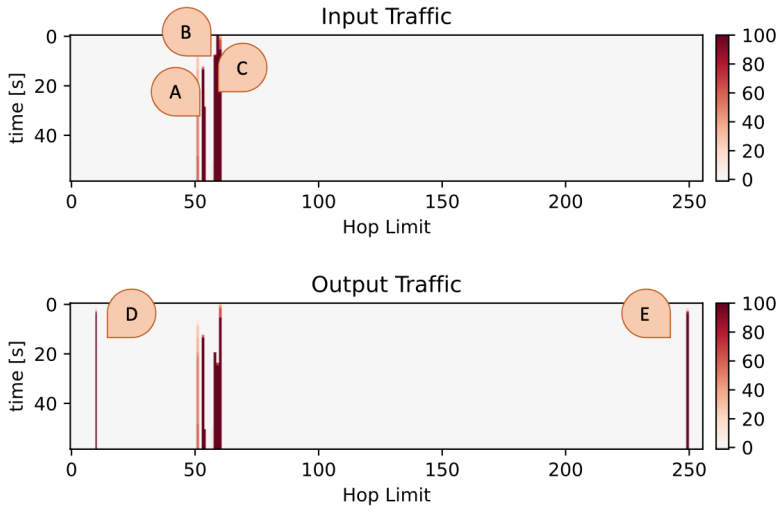

Fig. 6. Heatmaps generated before and after using pcapStego for creating covert channels in the Hop Limit.

traffic. Since original values of the Flow Label have been replaced with the secret data, the heatmap of the output traffic exhibits alterations of the corresponding bins. Specifically, the flows A and B are completely "exhausted" by the covert channel, whereas the conversation C is only partially used as a carrier. Moreover, the presence of hidden data leads to many values of the Flow Label not present in the original, input .pcap. This can be viewed in the map in terms of the increasing "heat" of some bins (denoted as "partial secret data", in the figure).

Similar considerations can be drawn when the covert channels implemented within the Hop Limit field. Specifically, Figure 6 depicts the heatmaps collected in our testbed. For the case of the original, input traffic, the map shows a clusterization around 64, i.e., the default value for the Hop Limit defined by the IPv6 standard implementation. Upon creating the covert channel, two different values become visible (denoted in the figure as D and E). Specifically, they correspond to 10 and 250, which are the values used by pcapStego to encode the secret information sent through the covert channels.

4.3 Performance of the Tool

The last round of tests aimed at investigating the performances of pcapStego. To conduct trials, we used a machine running Ubuntu 20.04 with an Intel Core i9-9900KF @3.60GHz and 32 GB RAM.

Table 1 summarizes measurements obtained for pcapStego running in interactive mode. Specifically, it reports the execution time of the main functions implementing the tool (i.e., the find_flows() and inject()), to inject a 110 byte long secret within the Flow Label field. To this aim, we considered different sizes of the .pcap file provided as the input. To have a fine-grained evaluation, we also performed tests with .pcap files composed of a different number of packets, especially to evaluate if the combination of both size and complexity of the traffic dump plays a role. As shown, the execution time needed by the find_flows() function to search for conversations able to contain the secret increases

.pcap file size [kbytes]	no. of packets [x10^3]	find_flows() [s]	inject() [s]
84	1	0.16	0.31
845	10	0.31	2.86
8,557	100	1.65	28.14
84,117	1,000	14.58	276.87
842,105	10,000	142.55	2,835.91

Table 1. Performance of pcapStego in interactive mode.

.pcap file size [kbytes]	no. of packets [x10^3]	read .pcap [s]	inject and write [s]
84	1	0.09	0.22
845	10	0.67	2.19
8,557	100	6.82	21.32
84,117	1,000	64.86	212.01
842,105	10,000	669.49	2,166.42

Table 2. Breakdown analysis of the inject() function in interactive mode.

with the size of the traffic trace. Yet, the required time remains bounded, especially for small- and medium-sized datasets. Instead, the inject() function turns out to be the real bottleneck of the processing stack implemented by pcapStego.

To better understand the limits of the inject() function, we performed further trials. Table 2 reports a breakdown of the operations performed by the function, i.e., reading the input .pcap (denoted as "read .pcap") and preparing the output traffic trace (denoted as "inject and write"). As shown, the performances heavily depend on the size of the dataset, which accounts for a major overhead for a .pcap file containing $10 \cdot 10^6$ packets. In this case, the preparation of the output .pcap requires ~35 minutes, which is not acceptable when using the tool for live demonstrations or for teaching. However, training duties seldom require to deal with such a large traffic capture. Besides, further measurements and a deeper analysis reveal that the injection/writing operations are limited by a twofold bottleneck: the I/O bound nature of read/write operations needed to process/produce the various .pcap files and the overheads caused by the Scapy stack for parsing and manipulating all the considered packets.

A similar evaluation campaign has been also done when pcapStego operates in bulk mode. To have a realistic reference usage pattern for the tool, we considered a varying population of secret messages to be sent through an equal amount of covert channels. Each secret message is composed of a randomly-generated string in the range of $8 - 64$ characters (e.g., a command for activating a backdoor or a sensitive information exfiltrated from a host). For this round of tests, we assess all the three embedding mechanisms implemented by pcapStego, i.e., we considered a balanced mix of covert channels targeting the Flow Label, Traffic Class and Hop Limit fields of the IPv6 header. Table 3 summarizes the obtained results.

Despite the number of channels, the execution time needed by the read_attack() function is always almost negligible, especially by considering that the tool is intended for preparing large-sized datasets in an offline flavor. Instead, the time needed by the find_flows() function increases mainly due to the need of building a larger set of 5-tuples representing the snapshot of the conversations within the input .pcap file. Similarly to the interactive case, the inject() function represents the most time-consuming building block of the tool. Specifically, when the number of channels to be created within the traffic trace approaches the $1,000$ unit, the handling of large .pcap files still accounts for reduced performances.

.pcap file size [kbytes]	no. of packets [x10^3]	no. of channels	read_attack() [ms]	find_flows() [s]	inject() [s]
845	10	10	0.23	0.36	3.04
845	10	100	0.28	1.36	3.25
845	10	1,000	-	-	-
84,117	1,000	10	0.21	15.07	304.9
84,117	1,000	100	1.22	18.37	327.72
84,117	1,000	1,000	10.53	106.72	471.55
842,105	10,000	10	0.26	140.31	2,452.69
842,105	10,000	100	1.34	155.95	2,611.58
842,105	10,000	1,000	10.73	470.57	3,743.88

Table 3. Performance of pcapStego in bulk mode.

.pcap file size [kbytes]	no. of packets [x10^3]	no. of channels	read .pcap [s]	inject and write [s]
845	10	10	0.67	2.37
845	10	100	0.66	2.59
845	10	1,000	-	-
84,117	1,000	10	65.03	239.87
84,117	1,000	100	66.34	261.38
84,117	1,000	1,000	66.23	405.32
842,105	10,000	10	661.27	1,791.42
842,105	10,000	100	665.55	1,946.03
842,105	10,000	1,000	655.27	3,088.61

Table 4. Breakdown analysis of the inject() function in bulk mode.

Again, we analyzed the breakdown of the timing behavior of the inject() function when used in bulk mode. Table 4 reports obtained results. In more detail, as the number of channels increases, the number of I/O-bound operations and invocations of the Scapy library increase as well, thus leading to inflated computational times (e.g., 3, 088.61 s when processing the largest .pcap file considered in our trials). Instead, as expected, the part of the tool processing the input traffic trace is quite insensitive to the complexity of the required embedding operations.

Lastly, we also evaluated the amount of resources used by pcapStego. For the sake of brevity, we limit our investigation to the interactive mode case. Results indicate that the memory footprint is proportional to the size of the .pcap to be processed. Specifically, the memory consumption ranges from 3.3 Mbytes to 24 Gbytes, for the smallest and largest considered traffic traces, respectively. This has to be mainly ascribed to the Scapy library having to process all the packets composing the .pcap. The size of the input also heavily influences the CPU usage. In fact, it is already equal to 93% when in the presence of medium-sized .pcap files (i.e., for a dataset composed of 100, 000 packets). Thus, the need of processing through Scapy large volumes of data both accounts for I/O and CPU bound operations impacting the overall performances.

5 CONCLUSIONS

In this paper we presented pcapStego, which is a tool for creating network covert channels within realistic traffic captures in .pcap format. As discussed, the main goals of the tools are to provide a simple playground for experimenting with network covert channels as well as to prepare datasets for automatizing and replicating experiments on information

hiding. Results indicated the effectiveness of the approach, especially to create `.pcap` snapshots to be replayed for testing real settings and appliances or to produce novel indicators, e.g., to support machine-learning-capable frameworks.

Future works aim at removing the limits of `pcapStego`. First, a relevant part of our ongoing research concerns expanding the tool to support a wider array of network covert channels, including those targeting IPv4 traffic. Second, relying upon third-part libraries and utilities surely reduces the implementation effort, but also accounts for some bottlenecks. Thus, future developments aim at improving the performances of `pcapStego`, for instance by developing ad-hoc functions in ANSI C or exploiting more efficient traffic analysis techniques/tools to process traffic captures. Lastly, we are also investigating the creation of more sophisticated transmission policies, for instance to support general signaling techniques among the covert endpoints.

ACKNOWLEDGMENTS

The project supporting this work has been removed for blind review.

REFERENCES

[1] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, and S. Zander. 2018. The new Threats of Information Hiding: the Road Ahead. *IT Professional* 20, 3 (2018), 31–39.

[2] B. Carrara and C. Adams. 2016. Out-of-band Covert Channels - A Survey. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–36.

[3] A. Carrega, L. Caviglione, M. Repetto, and M. Zuppelli. 2020. Programmable Data Gathering for Detecting Stegomalware. In *Proceedings of the 2nd International Workshop on Cyber-Security Threats, Trust and Privacy Management in Software-defined and Virtualized Infrastructures (SecSoft)*. IEEE.

[4] L. Caviglione. 2021. Trends and Challenges in Network Covert Channels Countermeasures. *Applied Sciences* 11, 4 (2021).

[5] L. Caviglione, M. Choraś, I. Corona, A. Janicki, W. Mazurczyk, M. Pawlicki, and K. Wasielewska. 2020. Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection. *IEEE Access* (2020).

[6] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli. 2021. Kernel-level Tracing for Detecting Stegomalware and Covert Channels in Linux Environments. *Computer Networks* 191 (2021), 108010.

[7] S. Floyd and V. Paxson. 2001. Difficulties in Simulating the Internet. *IEEE/ACm Transactions on Networking* 9, 4 (2001), 392–403.

[8] J. Fridrich, T. Pevnỳ, and J. Kodovskỳ. 2007. Statistically Undetectable jpeg Steganography: Dead Ends Challenges, and Opportunities. In *Proceedings of the 9th workshop on Multimedia & security*. ACM, 3–14.

[9] D. Gibert, C. Mateu, and J. Planes. 2020. The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges. *Journal of Network and Computer Applications* 153 (2020), 102526.

[10] J. Heidemann and C. Papdopoulos. 2009. Uses and Challenges for Network Datasets. In *2009 Cybersecurity Applications & Technology Conference for Homeland Security*. IEEE, 73–82.

[11] J.-F. Lalande and S. Wendzel. 2013. Hiding Privacy Leaks in Android Applications Using Low-attention Raising Covert Channels. In *2013 International Conference on Availability, Reliability and Security*. IEEE, 701–710.

[12] B. W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615.

[13] N. Lucena, G. Lewandowski, and S. Chapin. 2005. Covert Channels in IPv6. In *Int. Workshop on Privacy Enhancing Technologies*. Springer, 147–166.

[14] W. Mazurczyk. 2013. VoIP Steganography and its Detection - A Survey. *Comput. Surveys* 46, 2 (2013), 1–21.

[15] W. Mazurczyk and L. Caviglione. 2014. Steganography in Modern Smartphones and Mitigation Techniques. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 334–357.

[16] W. Mazurczyk and L. Caviglione. 2015. Information Hiding as a Challenge for Malware Detection. *IEEE Security & Privacy* 13, 2 (2015), 89–93.

[17] W. Mazurczyk, K. Powójski, and L. Caviglione. 2019. IPv6 Covert Channels in the Wild. In *Proceedings of the 3rd Central European Cybersecurity Conference*. 1–6.

[18] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho. 2019. A Survey of Network-based Intrusion Detection Data Sets. *Computers & Security* 86 (2019), 147–167.

[19] J. Saenger, W. Mazurczyk, J. Keller, and L. Caviglione. 2020. VoIP Network Covert Channels to Enhance Privacy and Information Sharing. *Future Generation Computer Systems* 111 (2020), 96–106.

[20] N. Sultana, N. Chilamkurti, W. Peng, and R. Alhadad. 2019. Survey on SDN Based Network Intrusion Detection System Using Machine Learning Approaches. *Peer-to-Peer Networking and Applications* 12, 2 (2019), 493–501.

[21] A. Thakkar and R. Lohiya. 2020. A Review of the Advancement in Intrusion Detection Datasets. *Procedia Computer Science* 167 (2020), 636–645.

[22] S. Zander, G. Armitage, and P. Branch. 2007. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Surveys & Tutorials* 9, 3 (2007), 44–57.