Victim-Aware Adaptive Covert Channels

Riccardo Bortolameotti[1(✉)], Thijs van Ede[1], Andrea Continella[2], Maarten Everts[1], Willem Jonker[1], Pieter Hartel[3], and Andreas Peter[1]

[1] University of Twente, Enschede, The Netherlands
r.bortolameotti@utwente.nl
[2] University of California, Santa Barbara, Santa Barbara, USA
[3] Delft University of Technology, Delft, The Netherlands

Abstract. We investigate the problem of detecting advanced covert channel techniques, namely *victim-aware adaptive covert channels*. An adaptive covert channel is considered victim-aware when the attacker mimics the content of its victim's legitimate communication, such as application-layer metadata, in order to evade detection from a security monitor. In this paper, we show that victim-aware adaptive covert channels break the underlying assumptions of existing covert channel detection solutions, thereby exposing a lack of detection mechanisms against this threat. We first propose a toolchain, Chameleon, to create synthetic datasets containing victim-aware adaptive covert channel traffic. Armed with Chameleon, we evaluate state-of-the-art detection solutions and we show that they fail to effectively detect stealthy attacks. The design of detection techniques against these stealthy attacks is challenging because their network characteristics are similar to those of benign traffic. We explore a deception-based detection technique that we call HoneyTraffic, which generates network messages containing *honey tokens*, while mimicking the victim's communication. Our approach detects victim-aware adaptive covert channels by observing inconsistencies in such tokens, which are induced by the attacker attempting to mimic the victim's traffic. Although HoneyTraffic has limitations in detecting victim-aware adaptive covert channels, it complements existing detection methods and, in combination with them, it can to make evasion harder for an attacker.

Keywords: Intrusion detection system · Network security · Covert channels

1 Introduction

Malicious software requires network communication in order to perform many of its functionalities [1]. For instance, bots receive instructions from Command and Control (C&C) servers [2], ransomware downloads encryption keys from remote locations [3], and information stealers exfiltrate data to external servers [4].

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
S. Chen et al. (Eds.): SecureComm 2019, LNICST 304, pp. 450–471, 2019.
https://doi.org/10.1007/978-3-030-37228-6_22

Malware activities last as long as security monitoring products do not identify the suspicious behavior. Consequently, attackers implement techniques to evade network monitoring tools to pursue their malicious activities for a longer period of time. These techniques are known as *covert channels* [5]. Attackers can in fact make their communication stealthier by applying advanced techniques that morph network messages into "benign-looking" traffic. We call this type of channels *adaptive covert channels*. Adaptation can be performed in two ways: *a priori adaptation* and *victim-aware adaptation*.

The most common type of adaptation is *a priori adaptation*, where the attacker implements her communication to look like benign software or protocol *before* compromising any machine. The anti-censorship community provides several examples [6,7]. Wright et al. [8] proposed to modify the packet sizes distribution of one class of applications in order to resemble another class. Moghaddam et al. [9] and Weinberg et al. [10] proposed two tools to camouflage TOR traffic as Skype and HTTP protocols, respectively. Examples can also be found in known malware samples, which mimic predefined, well-known applications, such as the Windows and Yahoo Messenger protocol [11], or common browsers.

However, there exist detection techniques that detect covert channels by modeling the normal network behavior of a monitored host (i.e., potential future victims) [12–14]. These techniques detect a priori adaptation because the choice of adapting to a specific application does not necessarily match the behavior of the victim. Thus, the adaptive covert channel would show different network characteristics from the victim traffic. An a priori adaptation technique can avoid detection if the detection model is known. In this setting, Fogla et al. [15,16] proposed polymorphic blending attacks (PBA), a technique to adapt shellcode payloads to fit the statistical representation of normal traffic embedded in the detection model. However, the detection model is not always available. In this work, we focus on more advanced techniques that evade detection without knowing the detection model details.

Covert channel detection systems mainly rely on two methodologies. *(1) Supervised learning* approaches analyze the characteristics of benign and malicious traffic to create a model that can reliably identify and distinguish characteristics between the two groups [17]. This approach is common and also effective, because the majority of covert channels preserve the same distinctive network characteristics in its messages. On the other hand, *(2) semi-supervised learning* approaches only rely on benign data, and the generated model describes the common characteristics of benign traffic [12,14]. Although this approach is usually less precise than the aforementioned one, it can successfully identify unknown covert channels, when their characteristics differ from benign data used for training. Nonetheless, both methodologies implicitly rely on the assumption that malicious traffic shows distinguishing patterns from benign traffic. *What happens if the malicious covert channel takes benign traffic characteristics of its victim into account to generate its messages?*

Victim-aware adaptation (VAA) occurs when malicious channels mimic the observed victim traffic to bypass detection. Casenove [18] proposed a technique

called polymorphic blending technique (PBT) that is based on PBA [15]. PBT learns the statistical representation of normal TCP payloads from the victim traffic. The data to be transmitted is then encoded using byte frequency distribution such that the payload statistical byte distribution is similar to the victim traffic. However, PBT does not preserve the correct syntax of the application layer protocol, including its metadata. By substituting bytes in the payload, the syntax of the messages may be disrupted. Similarly, also PBA does not try to resemble a message syntactically similar to the victim traffic, but focuses on the byte frequency distribution. Consequently, detection systems that rely on application layer information for detection (e.g., [12,14,17]) could detect PBT due to corrupted messages (e.g., unparsable syntax) or unrecognizable byte sequences. Yarochkin et al. [19] propose to use covert channels over different protocols depending on the protocols used by the victim. Although their solution adapts to the victim, they do not consider to mimic the victim messages, thus their content would show deviating characteristics from the victim traffic and the covert channel would be detected.

In this work we investigate the ineffectiveness of state-of-the-art covert channel detection mechanisms against VAA covert channels, where the attacker mimics the syntax of the victim messages. Although we are not aware of malware families using VAA covert channels, there exist malware families capable of sniffing and manipulating their victim traffic [20,21]. Moreover, we explore a deception-based technique to detect VAA covert channels, which relies on different assumptions than existing detection method. Our technique is complementary to existing detection methods. The combination of our proposed technique and existing detection solutions provides better protection against VAA covert channels than existing solutions used as standalone techniques.

As part of our investigation of VAA covert channels, we introduce Chameleon, a toolchain to generate synthetic datasets containing data exfiltration attacks over VAA covert channels. Chameleon generates exfiltration attacks by mimicking legitimate traffic at the application layer. Then, we use the synthetic dataset to evaluate three HTTP-based detection techniques [12,14,17], which are based on semi-supervised [12,14] and supervised learning [17]. We focus the efforts of our evaluation on HTTP, because a vast majority of malware implements its communication channels over HTTP [1,22]. Moreover, from the attacker's point of view, the evasion techniques should work even in case a defender has full access to the plaintext data (e.g., via a TLS-proxy). Our evaluation shows that existing approaches are not suitable for detecting VAA data exfiltration attacks, because these attacks break the underlying assumptions of the detection methods. The results show that existing solutions either do not detect the attack or have a false positive rate above 7%, rendering them impractical in practice. Finally, we propose HoneyTraffic, a deception-based detection against VAA covert channels. HoneyTraffic consists of a client-side component that generates traffic containing *honey tokens*. When an advanced attacker mimics the victim traffic, she ends up mimicking honey tokens, which can in turn be easily detected. This produces inconsistencies that can be detected by

our monitor. Following [23], there are no comparable network-based deception techniques. We evaluate our approach against the adaptation strategies implemented by Chameleon in a worst-case scenario. The results show that HoneyTraffic can detect specific strategies of VAA covert channels. HoneyTraffic makes it harder for VAA attackers to evade detection, but not impossible.

Our main contribution lies in the implementation of Chameleon and the evaluation of existing detection solutions. Code and dataset will be made publicly available. VAA covert channels are a serious threat that is hard to detect. HoneyTraffic is a first attempt to their detection and it allows for the detection of some, but not all, VAA covert channels, thereby making attackers evasion more difficult. However, VAA covert channels remain a threat hard to detect.

2 Chameleon

The main goal of our work is to evaluate the effectiveness of VAA covert channels against existing detection solutions that use benign traffic to learn detection models. For this purpose, we present Chameleon, a toolchain that generates traffic samples containing data exfiltration attacks over VAA covert channels. We use Chameleon to evaluate existing network-based detection solutions. We introduce Chameleon since we are not aware of malware samples using VAA covert channels.

The core observation behind Chameleon is the following. Detection solutions, both supervised and semi-supervised, often learn detection models from the network characteristics of (non-compromised) machines. The network traffic of a machine is generated by the applications installed on it. Thus, Chameleon can generate network messages that mimic their application-layer metadata (e.g., using same headers and values), and then it can secretly hide data in the mimicked messages. Since such messages share the similar characteristics as those used to learn the detection model, the security monitor will not be able to effectively detect the covert channel. In other words, Chameleon fits the detection model by mimicking the victim traffic, similarly to PBA [15]. However, Chameleon does not require access to the detection model and focuses on the syntax of messages and not their byte distribution.

2.1 Threat Model

We assume the attacker establishes a hidden communication channel to bypass a security monitor. The attacker controls both client and server, and the monitor can read all communications as plaintext (e.g., via a TLS proxy [24]). The attacker can observe the network traffic of the compromised host and mimics its network messages. Hence, we assume the attacker has enough privileges on the victim to read its network interface. Although this requirement is not typically trivial to satisfy for an attacker, there are attackers with such capabilities [20,21]. This does not imply that the attacker has system privileges on the compromised machine. This can occur, for instance, when sniffing tools (e.g., `tcpdump`) are

given the permission and capability to allow raw packet captures (e.g., `setcap cap_net_raw,cap_net_admin`). Lastly, in this work we assume that the attacker only adapts outgoing traffic. We analyze a unidirectional VAA covert channel, because it is enough to bypass existing detection systems.

2.2 System Overview

Chameleon takes a network trace as input and embeds in it the traffic of a VAA covert channel. The new trace is used to evaluate detection systems, because it contains normal traffic, from the original trace, and the exfiltration of data over a VAA covert channel.

Chameleon works in two steps. The first step is *Adaptive Traffic Generation*, which involves a client and a server. The client follows two alternating phases: during the *collection* phase it reads the network trace provided as input, and during the *blending* phase it generates the adapted network messages. This terminology was introduced by Casanove [18]. Then, the adapted messages are sent to the server, which responds to the client. The second step is *Traffic Integration*, which involves a set of tools used to modify network traces. These tools are used to integrate the adapted traffic, stored in a temporary trace, into the original trace. Currently, Chameleon is implemented to work for HTTP traffic.

We decided to design Chameleon as a "dataset generator", so we could generate datasets including VAA covert channels traffic by using only network captures. This allows us to obtain datasets from machines running different operating systems and applications, and to easily test defensive mechanisms in different settings. This makes it also easier for other researchers to generate VAA covert channel datasets without requiring a dedicated machine on which Chameleon needs to run. Instead, previously captured network traffic can be injected with adapted traffic to analyze.

2.3 Adaptive Traffic Generation

As in any covert channel, client and server need to share a set of parameters before communicating in order to identify the hidden data from the messages. Moreover, an adaptive client may use a set of message parameters to specify its way of communicating. Below we discuss the setup parameters, message parameters and the client implementation. Since our work focuses on unidirectional VAA covert channels, where only the client adapts to the victim, Chameleon implements the server as a listener that returns a default response.

Setup Parameters. *SP are shared* between the adaptive client and server before any communication is established. These parameters provide the basic information for the client and the server to identify the hidden data within the network messages. They are defined as $SP = \langle p, e \rangle$. For each network protocol p, the adaptive client and the server share a list of encoding algorithms e. The parameter e also contains a delimiter to identify where the hidden data "starts".

Message Parameters. *MP are not shared* between client and server. They are used to specify how messages should be crafted in the collection and blending phases such that they appear to be legitimate messages, while hiding secret information. In this work, MP are only set at the client side. Here $MP = \langle c, s, l, i, b, a \rangle$, where c is the timeout of the collection phase; s is the maximum size of the data to embed in each message (i.e., bit-rate of the covert channel); l is the location where the exfiltrated data is embedded (e.g., URI, headers, body); i is the delay between sending out messages; b is the timeout for the blending phase; and a is the type of application the attacker wants to mimic (e.g., Firefox).

Client Implementation. Figure 1 gives an overview about the process generating adapted messages in Chameleon. During the collection phase, Chameleon selects an application a to mimic. All messages from a are collected based on the `User-Agent` field. Chameleon stores the list of header fields and the values associated to each header in the *header set* and *header-value dictionary*, respectively. The collected information and the data that needs to be transmitted (e.g., file to exfiltrate) are passed to the blending phase, which encodes the data using scheme e and splits the file into chunks of size s, obtaining a set of *data items*. Before each data item is transmitted, a template is generated. A template represents a set of headers, and their associated values, that aim at mimicking the victim traffic. An example of template is shown in Fig. 1 with green color font. Headers are chosen randomly from the header set, while the associated values are randomly chosen from the header-value dictionary. The data item to transmit (i.e., the covert message) is inserted in the location, within the template, indicated by the parameter l (highlighted in red in Fig. 1), thereby generating the adapted message. Thus, Chameleon *inserts data in a single message location*, in order to maximize the mimicked parts of the message. Finally, the adapted message is sent out to the server using interval i. After b seconds, the process restarts from the collection phase to mimic the most recent traffic. The server retrieves the message by identifying the delimited described in the setup parameter e and extracting the adjacent data.

Our implementations uses HTTP and base64 as setup parameters. Message parameters MP can be set before running the client and the server. Please note that whoever establishes a covert channel is also in control of both client and server behavior. Therefore, *an attacker may use the protocol in a way that is semantically wrong, but she can still communicate successfully.* For example, the client can generate HTTP GET requests for resources that do not exist on the server, but the server may still provide a valid response (e.g., 200 OK).

2.4 Traffic Integration

The second step Chameleon performs is the integration of the VAA traffic, observed between client and server, into the original trace provided as input. First, the traffic of the VAA channel is captured using `tcpdump`, and it is stored into a temporary network trace. Then, the IP addresses in the temporary trace

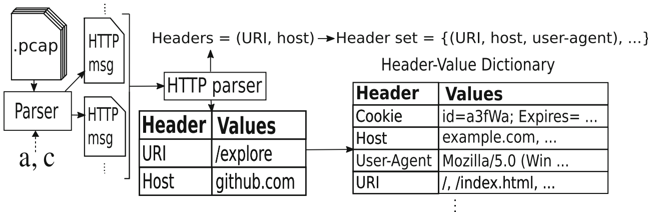
are rewritten according to the victim's IP contained in the original trace using `tcprewrite`. Similarly, the timestamps are rewritten in order to fit the time period of the original trace, and we achieve this using `editcap`. Finally, now that both traces are consistent in terms of IP address and timestamps, we merge them using `mergecap`. These tools are part of the Wireshark suite. The final trace contains labeled victim's traffic and the attack, thereby being ideal for an evaluation of detection systems.

3 Experimental Evaluation

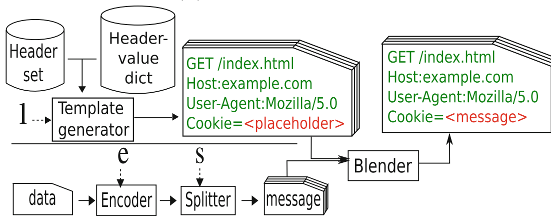
3.1 Dataset

For the generation of the dataset, we need network traces of benign traffic representing different hosts, and thus, potential victims. We choose the dataset published by Sharafaldin et al. [25] for this purpose. The reasons behind this choice are the following: (1) it contains traffic from multiple hosts over a time period of a week, allowing defenses to be trained with the traffic of the beginning of the week, and to be tested on the rest; (2) it contains traffic emulating user behavior and it contains different type of machines (e.g., servers and workstations); and (3) it is publicly available, so our generated dataset can be publicly released, without compromising any user privacy, making our work easier to be reproduced.

We extracted from the dataset of Sharafaldin et al. [25] all the outbound traffic generated by 9 different hosts. For each host, we obtained 5 days of network



(a) Collection phase



(b) Blending phase. Green text represents the template, while red text represents the covert message.

Fig. 1. Overview of the VAA covert channel implementation by CHAMELEON. (Color figure online)

Table 1. Mimicking strategies applied in our dataset.

Host ID	file [KB]	c [s]	s [B]	l	i [s]	b [s]	a
5	500	5	128	H	0.00	∞	M
8	50	5	0–128	U	0–5.00	∞	B
9	50	5	128	B	0.00	5	M
12	500	5	128	H	0.00	∞	N
16	50	5	0–128	U	0–5.00	∞	N
17	500	5	0–128	B	0–1.00	∞	B
25	500	5	0–128	U	0–0.05	5	B
50	500	5	0–128	H	0–0.05	5	B
51	500	5	0–128	B	0–0.05	5	B

traffic (from Monday to Friday), where each day of traffic is a single network trace. For each host, we choose a different set of message parameters MP, which represents a *strategy* a VAA covert channel may use to communicate. We ran CHAMELEON using each host's network trace and defining the MP according to the strategy assigned to the host. Overall, the dataset resulted in 45 pcap traces containing both our VAA covert channels and hosts traffic. The dataset contains 16.6 GB of network traffic, of which 476 MB (120k HTTP requests) originate from the adapted communications.

Table 1 lists the message parameters of each strategy applied to each host. The column *file* indicates the size of the file we transmitted during the adaptive communication. The values of l are denoted by B for body, H for a random header, U for URI parameters and M for a random choice between header and URI. While the values of a are given as B for browser, N for non-browser, or M for a mix of both. The strategies represents different exfiltration scenarios. Such scenarios include data exfiltration of files large 50 Kb and 500 Kb at different speeds, exfiltration locations, and application types. For instance, strategies for hosts 8 and 16 wait up to 5 s between sending messages, whereas strategies for hosts 5, 9, and 12 do not wait between attempts at all.

Existing Detection Solutions. The three detection solutions are: DUMONT [14], DECANTeR [12], and HED [17]. We selected these tools from the state-of-the-art for three reasons: (1) their detection models rely on HTTP traffic characteristics that include payload and headers, and (2) their detection models leverage benign traffic information; and (3) they cover both semi-supervised (DUMONT and DECANTeR) and supervised learning (HED). The implementation of DUMONT and DECANTeR are available online, while we obtained the code of HED from the authors.

DUMONT generates a detection model for each monitored host, and the model is a One-class SVM that describes the HTTP network characteristics according to several numerical features representing different location of HTTP requests (e.g., headers, URI, Body). DECANTeR models the traffic of each mon-

itored host by fingerprinting installed applications after observing their traffic, and the fingerprints describe different network features of the application. HED generates a binary SVM classifier given two sets of traffic, representing "normal" and covert channel traffic, respectively. The binary SVM relies on more than a thousand features related to HTTP requests to create the model. We refer the reader to the original works for additional information on these systems.

Evaluation Setup. We evaluate the classification performance of each detection tool in terms of *accuracy* (ACC), *false positive rate* (FPR), *true positive rate* (TPR), *detected attack* (DA) and *detected strategy* (DS). An attack is detected if at least one HTTP request is triggered as malicious within a single traffic sample. A strategy is detected if at least one HTTP request is flagged as malicious within *all* the four traffic samples of that specific strategy. We compute the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), for the whole dataset.

The detection systems has been evaluated as follows. For DUMONT, we first train a model for each host using the first half of the benign training data of the Monday sample. Then, we calibrate the DUMONT model using the same amount of malicious and benign data present in the second half of the Monday sample[1]. We test the remaining four samples, from the same host, against the model and we collect the results. Similarly, for DECANTeR we train the models using the benign traffic of the Monday sample. Then, we test the remaining four samples for each host and we collect the results. For HED we train a single model for all hosts using their Monday samples, and we evaluate it against the remaining samples. However, if we generate a supervised model from a set of known VAA malicious samples, we train the model to detect the malicious samples in the context of their specific victims. Hence, *it is not guaranteed that the same samples are effectively detected in the future*, because their traffic may look different if they affect other victims. Thus, we evaluate HED in two different scenarios representing two deployment assumptions. The first scenario is HED_{pk}, assuming HED has *partial knowledge* of the malicious traffic, thus the training model contains all benign host traffic but only a subset of malicious traffic. In our evaluation we randomly excluded four hosts' malicious traffic (i.e., host 50, 12, 8 and 25 in Table 1). The second scenario, HED_{tk}, assumes HED has *total knowledge* of the malicious hosts' traffic, so the training model contains all benign and malicious traffic. HED_{pk} is a more realistic scenario, because it assumes that the defender does not know all the characteristics of a VAA attacker beforehand.

3.2 Results

We consider a solution to be effective if it has a true positive rate higher than 70% and a false positive rate lower than 2%. A high TPR means that it is unlikely that the attack is going to be missed. Thus, in practice a low TPR can be very costly for a company. A low FPR means the detection solution rarely triggers

[1] Please note that in the original work [14] there is no clear guidance about the quantities of data needed for the calibration step.

false alarms. According to Sommer and Paxson, in their seminal work about IDSs and machine learning in real environments [26], limiting false positives must be a top priority of any IDSs. Table 2 shows the overall detection performance of the evaluated detection solutions for all the 9 strategies. Table 3 shows a detailed overview about detection performance per strategy.

Semi-supervised Learning. DECANTeR and DUMONT are not effective in detecting VAA covert channels, as shown in Table 2. The main reason is that adaptation allows the malware to camouflage its characteristics with those of the victim. Thus, the malware fits the trained model and breaks the fundamental assumption of anomaly detection, namely that malware shows different patterns than benign traffic.

DECANTeR is not effective against CHAMELEON because its classification system is based on specific associations of *some* header fields and values. Since CHAMELEON uses existing associations of header fields and values in its template, DECANTeR considers adapted messages as generated by installed applications. CHAMELEON can exfiltrate data through the `URI`, `Body`, `Accept`, and other headers, without being detected, as long as the "key" header values (i.e., `Host`, `User-Agent`, `Accept-Language`) match the fingerprint values. If data is hidden within such "key" headers, then DECANTeR likely detects the adapted traffic. The most successful strategies involved the mimicking of background applications. The reason behind is that adapted messages had the same headers sequences, same `Host` values and `User-Agent`, which are three fundamental features for background application fingerprinting in DECANTeR. By adapting to the victim's traffic, CHAMELEON completely avoided detection in two cases (strategies 12 and 16), as shown in Table 3. DECANTeR also performs poorly for strategies 5 and 9, when the malware randomly adapts to different applications.

DUMONT is not effective against CHAMELEON because its classification system mostly relies on statistical features describing the length and structure of different parts of a message (i.e., headers, `URI` and `Body`), their entropy, and the average length of header and `URI` values. The template created by CHAMELEON contains data with similar statistical characteristics, because it comes from the same applications. Thus, the data contained in the template help hiding adaptive traffic from detection. Therefore, if CHAMELEON also hides data in locations that typically contain large amounts of data such as `Body` and `URI`), then CHAMELEON is likely to fit the trained length characteristics learned by the model, and it avoids detection. Overall, DUMONT misses almost a third of the samples. It misses all the samples for strategies 16 and 9, likely due to the small amount of data being exfiltrated. Although DUMONT detects some malicious requests for strategies 5, 8, 25 and 50, the vast majority of malicious requests are still not detected. This justifies the low accuracy.

Supervised Learning. HED yields the best results in terms of detection. All the samples are detected. The extra attack knowledge used to create the model helps HED to improve its performance. However, the FPR for both scenarios is 2 or 3 time higher than for other solutions, making HED not effective against CHAMELEON. In the original paper [17], HED showed a FPR of 0.01% against

non-VAA covert channels. In our evaluation *the FPR increases two orders of magnitude*. The reason behind the high FPR is the similar statistical representations of benign and adapted *messages*. Templates contain data statistically very similar to the one used in training, because it is data from the same application. Thus, when the model is generated, it is difficult to find a decision function that can reliably separate the two classes of traffic. Moreover, due to high number of requests observed in HTTP traffic and the inherent heterogeneity of HTTP, HED often misclassifies benign HTTP traffic as malicious. Table 2 shows that HED in its ideal setting HED_{tk}, where all strategies are known during training phase, achieves 96% accuracy. The accuracy is not 100% because HED_{tk} generates FPs due to the similarities between benign and malicious messages. However, in the more realistic setting HED_{pk}, the accuracy is 89%, where only some strategies are known at training time. This 7% accuracy drop is due to the lack of knowledge about new adaptive strategies. As shown in Table 3 the true positive rate is lower for strategies 8, 25, and 50, which were excluded from the training dataset.

Lessons Learned. *Our evaluation shows that existing defensive mechanisms are not effective against detecting VAA covert channels*, because they cannot effectively detect them while, at the same time, triggering few false alerts. Semi-supervised learning solutions are not effective against VAA attackers, because malicious traffic fits the model of benign communication, and thus, malicious connections are not flagged as anomalies. Supervised learning solutions are not effective for two reasons. First, the classes in the training dataset are represented by similar sets of data, thus there is not a clear distinction between the two classes features, which is a fundamental requirement for supervised approaches to be effective. Second, the VAA covert channel data used during the training of the system may not be representative for successive attacks, since the attacker mimics the traffic of new victims, which may show different network patterns. Both issues are present in HED: the first is highlighted by the high FPR, while the second is described by the accuracy drop between the two HED scenarios HED_{tk} and HED_{pk}.

Limitations. Chameleon has three main limitations. First, it only mimics the content of network packets and not the interaction between client and server. A

Table 2. Overall performance of existing defenses in terms of accuracy (ACC), false positive rate (FPR) and detected attacks (DA).

Existing defense	Performance			
	Acc	TPR	FPR	DA
DUMONT	49%	10%	2%	17/35
DECANTeR	70%	50%	3%	12/35
HED_{pk}	89%	85%	7%	35/35
HED_{tk}	96%	93%	7%	35/35

Table 3. Overall performance of DUMONT, DECANTeR, HED_{pk} and HED_{tk} per strategy, in terms of true positive rate (TPR), false positive rate (FPR) and detected strategies (DS) (× represents a missed strategy).

Strategies	DUMONT			DECANTeR			HED$_{pk}$			HED$_{tk}$		
	TPR	FPR	DS	TPR	FPR	DS	TPR	FPR	DS	TPR	FPR	DS
5	0.001	0.01	✓	0.50	0.02	✓	0.99	0.06	✓	0.50	0.02	✓
8	0.005	0.01	✓	0.18	0.05	✓	0.50	0.05	✓	0.99	0.07	✓
9	0.000	0.01	×	0.09	0.05	✓	0.98	0.07	✓	0.99	0.07	✓
12	0.133	0.03	✓	0.00	0.00	×	0.99	0.07	✓	0.99	0.05	✓
16	0.000	0.01	×	0.00	0.03	×	0.98	0.07	✓	0.99	0.07	✓
17	0.371	0.03	✓	0.51	0.03	✓	0.99	0.10	✓	0.98	0.09	✓
25	0.023	0.01	✓	0.61	0.02	✓	0.55	0.12	✓	1.00	0.10	✓
50	0.001	0.03	✓	0.78	0.31	✓	0.68	0.04	✓	0.93	0.14	✓
51	0.150	0.06	✓	1.00	0.03	✓	0.99	0.10	✓	0.99	0.05	✓

detection mechanism that models client-server interactions may effectively detect Chameleon. However, web traffic is heterogeneous and inconsistent client-server interactions may be frequent, considering the large volumes of web traffic. Thus, designing an effective heuristic is non trivial. Second, Chameleon does not implement the concept of connection state. However, this limitation does not affect Chameleon over HTTP, because it is challenging to reliably monitor, for detection purposes, HTTP connection states (i.e., Cookies) due to the high heterogeneity of their usage and values across different web services. Third, Chameleon can be detected using signatures or dedicated heuristics to identify specific patterns in the tool implementation. A simple example is to create a signature to the static server response. As discussed by Houmansadr et al. [6], this is always possible for defenders to detect mimicking techniques. However, these detection approaches are easy to evade since they rely on implementation details rather than on the underlying patterns of the exfiltration technique.

Finally, it may seem trivial to detect Chameleon using heuristics that identify replicated content in network messages. However, this is not the case due to some challenges that defenders must take into account: (1) benign applications regularly generate similar messages over time (e.g., scripts uploading data or downloading dynamic content), (2) defenders would have to monitor large quantities of data and keep a detailed historical record for each host; and (3) Chameleon is a tool that can be configured with different parameters (e.g., increase delay to enforce larger historical analysis for defenders). These challenges makes it non-trivial to effectively detect VAA covert channels using heuristics that rely on passive network traffic analysis.

4 Honey Traffic

We introduce the concept of HoneyTraffic, a deception-based mitigation against VAA covert channels. We propose to *turn the table on the attacker* and make the challenging task of detecting VAA covert channel her problem, while we use her offensive techniques for the purpose of detection.

The intuition of HoneyTraffic is that we can generate network messages that mimic existing applications and, at the same time, contain secret tokens. Thus, the attacker, while adapting to the victim's traffic, includes such tokens in its messages. A security monitor, which is aware of the tokens and knows how to identify them, can detect the presence of messages containing tokens generated by "unknown" clients, thereby detecting the VAA covert channel.

The goal of HoneyTraffic is to detect the class of VAA covert channels represented by Chameleon, where the attacker mimics existing messages and embeds the secret data to exfiltrate in one of the message locations. HoneyTraffic is not intended to detect all possible types of VAA covert channels. More importantly, HoneyTraffic *is not intended to detect covert channels that do not adapt to the victim.* Such threat scenario is already covered by existing approaches (e.g., [12,14,17] for HTTP). Hence, we consider HoneyTraffic to be a *complementary* solution to existing detection approaches. As other deception-based techniques, such as honeypots, stack canary or canary tokens, HoneyTraffic relies on the assumption that an attacker cannot distinguish between fake and real items. Although canary and honey tokens are conceptually similar, they protect different aspects of the information system. Canary tokens trigger alerts if decoy files are accessed, while HoneyTraffic triggers alerts if someone copied network messages. In case an attacker accesses common files (i.e., not decoys) and exfiltrates them over an adaptive covert channel, canary tokens do not trigger alerts, while HoneyTraffic can.

Assumptions. We assume there exists a *honey client* installed on each machine (i.e., potential victim). The honey client generates network messages mimicking the machine's traffic. In other words, the honey client establishes VAA communication channels. Moreover, we assume there exists a *honey server* and a security monitor (e.g., next-generation firewall or NIDS). The monitor is usually already part of the company's infrastructure. The honey server can establish secure communications with both the honey client and the security monitor to provide the setup information. As discussed in Sect. 2.1, VAA covert channels, such as Chameleon, assume a strong defenders capable of accessing all communication in plaintext. Thus, we assume HoneyTraffic to be capable of accessing the communication in plaintext (e.g, using a TLS proxy).

We assume the attacker compromises a machine where a honey client is installed, and we assume the attacker is aware of the presence of honey messages. However, the attacker does not know the list of honey tokens. We consider the attacker to be not capable of running advanced detection heuristics on the infected machine to identify the presence of honey tokens. It is not realistic

to assume an attacker can run advanced detection heuristics like a host-based intrusion detection, while trying to hide its presence.

4.1 System Overview

HoneyTraffic is composed by three main components: a client, installed on each monitored host, a server and a security monitor. We refer to the client and server as *honey client* and *honey server*, respectively. The system works in two alternating phases: a setup phase and a detection phase. The setup phase is responsible for delivering the information necessary to run the detection mechanism to each component. During the detection phase, the system monitors the network to spot adaptive covert channels.

Setup Phase. The *honey server* is the main component during the setup phase. It is responsible for the generation of all the information needed to run the detection system. The honey server generates a set of *honey tokens*, a set of random IP addresses, the setup and message parameters (SP and MP), and a set of *honey signatures* (e.g., regex) to identify the tokens in the network traffic. The random IP generation process excludes existing network nodes that may trigger network errors from existing machines using the chosen IPs, or by network devices that know these IPs do not exist. This precaution is needed to avoid attackers being able to easily identify honey traffic. Then, the server securely communicates the tokens, IP addresses, and setup and message parameters to the honey client, while it sends the signatures to the security monitor.

The setup phase reoccurs after a specified time period to *substitute old tokens and signatures with new ones*. In other words, the system updates regularly its key detection material. In case the setup phase is repeated after a short time (e.g., a day), it becomes difficult for an attacker to identify honey tokens.

Honey Client Communications. Upon receiving the information from the server, the *honey client* starts collecting information about the machine's traffic (collection phase). Once enough information is collected, the client creates a template for its network messages that resembles the machine's traffic, and it embeds the honey tokens within the template (blending phase). In other words, the honey client follows the same process of the VAA covert channel client discussed in Sect. 2.3. In this process it uses the SP and MP parameters provided by the honey server. Lastly, the client spoofs the destination IP of its messages using one of the addresses provided by the server, and it sends the messages.

Detection Phase. Once the security monitor receives the signatures from the honey server, it starts monitoring the network traffic. Whenever the security monitor identifies a honey token (*signature hit*), it redirects the message to the honey server for further inspection. Assuming the tokens are unique strings generated by the honey client, which is a common assumption in signature-based detection, a signature hit can be triggered for two reasons: either the token was included by the honey client or from a malicious adaptive application. If the message has as destination one of the random IP addresses generated during the

```
GET /pypi/v/dpkt.svg HTTP 1.1

Host: img.shields.io
User-Agent: Mozilla/5.0 ... Firefox 54.0
Accept: */*
Accept-Languate: en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie:__cfduid=dce101b3e09a9a09d5dbf
2afed17dcdc71413792427910
Connection: keep-alive
```
a)

```
GET /pypi/adlogin_page/v/dpkt.svg HTTP 1.1

Host: img.shields.io
User-Agent: Mozilla/5.0 ... Firefox 54.0
Accept: */*
Accept-Languate: en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie:__cfduid=dce101b3e09a9a09d5dbf2afe
d17dcdc71413792427910; timestamp=13 Nov
2018; 10:00
Connection: keep-alive
```
b)

```
GET /pypi/adlogin_page/v/dpkt.svg HTTP 1.1

Host: img.shields.io
User-Agent: Mozilla/5.0 ... Firefox 54.0
Accept: */*
Accept-Languate: en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie:__cfduid=RXVybyBJRUVFIFMmUCAyMD
E4IFN1Ym1pc3Npb24h
Connection: keep-alive
```
c)

Fig. 2. Example of HoneyTraffic. (a) represents an example of a message from an existing application. (b) depicts a honey message, which mimics an existing application but also embeds the honey tokens (in red). (c) shows a malicious message that adapts to the victim's traffic, which includes its secret data (in blue) and an undetected honey token (in red). (Color figure online)

setup phase, then the message is generated by the honey client. Otherwise, the message is generated by an adaptive application. Since the attacker's goal is to communicate with servers she controls, it is very unlikely one of her destinations matches with one of the randomly generated IP addresses.

In case the message is identified as originating from the honey client, the server answers with a standard response. It is important that the responses are not static and look like legitimate traffic, to avoid that the attacker easily identifies honey communications. The response is needed because the IP is spoofed, and a real destination is not reached. In case the message is considered malicious, the server triggers an alert.

4.2 Example of Honey Traffic

Let us assume the honey server generates three different tokens: (1) "/adlogin_page/", as a subfolder of a `URI` path; (2) "timestamp: 13 Nov 2018; 10:00", as a `COOKIE` parameter; and (3) "686897696a7c976b7e", as a `ETAG` value. The honey server also generates: (i) a list of fake destination IP addresses [192.168.1.100, 23.45.21.32, 142.59.23.1]; and (ii) signatures for a NIDS to identify each honey token. For instance, using Snort syntax, the following can be a signature for the first token: *alert tcp any any -> any 80 (msg: "HONEY token 1"; content: "/adlogin_page/"; http_uri; sid:2000001;)*. The honey tokens, signatures and list of fake destination IPs are shared with the honey client and the NIDS.

Now assume that during the *detection phase* the honey client mimics a message of an existing application (Fig. 2a), and it embeds two different honey tokens in the message. The message is then sent over the network to the fake address: 23.45.21.32. Figure 2b shows how the message generated by the honey client may look like. Next, the security monitor, which analyzes each single message, finds two signature matches with the honey client, one per token. The monitor forwards the message to the honey server, which verifies that the destination IP is part of the set of random destination IPs generated during the setup phase. Hence, the message is considered to be generated by the honey client.

Now let us assume a malware uses an VAA covert channel to communicate with its server with IP 123.23.67.97. The malware sniffs the victim's traffic, and it observes the messages shown in Fig. 2a and b. It generates an adapted message, and it embeds the data to exfiltrate in the `Cookie` (see Fig. 2c), and it sends it to 123.23.67.97. In this scenario, the security monitor finds one signature, because the token in the `URI` (i.e., "/adlogin_page/" in red in Fig. 2c) is present in the message. The monitor forwards the message to the honey server, which marks the message as malicious because it has a destination address that does not match with those created during the setup phase.

4.3 Generating Honey Tokens

HoneyTraffic is an effective defensive mechanism against VAA attackers, if the honey tokens cannot be reliably detected by an attacker. Thus, a honey token must be a sequence of bytes that is unlikely to appear in traffic generated by other applications. These sequences are not difficult to generate.

Although there are several possibilities to generate tokens, the best locations are those headers that are commonly used, but are often associated with different values. As analyzed by Borders and Prakash [27], a fixed fraction of HTTP connections contain high entropy data. Thus, *it is difficult even for an advanced attacker to discriminate whether the header values contains tokens or just benign traffic*. The headers that are commonly used in HTTP and have such properties are: `Host`, `Cookie`, `Referer`, `Body` and `URI`. For example, for `Host` we can generate a fake subdomain, or for the `Referer` we can generate a fake URL that was never requested before. These values are likely unique in the traffic. Other headers that can be used for honey tokens are: time-based headers (e.g., `If-Modified-Since`, `If-Unmodified-since`, `Date`), where the tokens are represented by timestamps, or headers related to check resources updates (e.g., `If-Match`, `If-None-Match`, `ETAG`), where the tokens are represented as random sequences of alphanumeric characters.

Honey tokens can be automatically generated by the honey server. *It is important that the server is aware of the expected syntax for the header value*, to avoid that malware notices simple format inconsistencies. Following the aforementioned examples, the server can use English words as subdomains to create `Host` tokens or as string in the URI path (e.g., Fig. 2b). Considering the large set of choices a defender has to generate honey tokens, the fact that tokens can be generated to be nearly indistinguishable from normal traffic, and the limited detection capabilities of the attacker, *it is unlikely the attacker can consistently detect honey tokens*. She would have to correlate the content of many messages, and identify patterns within a limited amount of time, because the setup phase periodically introduces new tokens. To complicate the detection even further, the honey client can use each token only once (i.e., one-time tokens), reducing the chances for the attacker to identify string patterns. In our analysis in Sect. 4.4 we discuss how much one-time tokens costs to the honey client. A single attacker mistake in judging the presence of a honey token costs her the detection.

4.4 Evaluation of Honey Traffic

HoneyTraffic relies on the assumption that, sooner or later, an attacker adapts to messages including honey tokens. Thus, the detection of VAA covert channels is not deterministic but probabilistic. In order to evaluate HoneyTraffic, we make the following assumptions: (1) the attacker and the honey client mimic the traffic of a browser application, namely the *target application*; and (2) the honey client hides honey tokens into a fixed number of header values.

Attackers are more likely to adapt to browser traffic, because browsers generate a lot of traffic, thus it is easier for the attacker to hide. Moreover, browsers represent the worst-case scenario for HoneyTraffic in terms of traffic overhead. The large volumes of browser traffic force honey clients to generate more traffic to detect a VAA attacker. Although we analyze HoneyTraffic for a specific application, in practice, HoneyTraffic should be generated for all communicating installed applications.

Applications can use different sets of headers during their communication. However, there is always a subset of headers that is used in all applications' network messages [12]. Thus, a honey client can use such headers to hide tokens in it. For example, browser traffic always contains a `URI` (i.e., needed to retrieve a web resource) and a `Host` value (i.e., domain to contact). Both headers contain data that changes very often, making it hard for an attacker to detect potential tokens. For this reason, we believe it is realistic to assume that a honey client can hide tokens in a fixed set of headers. Specifically, following the aforementioned example, we assume the honey client hides tokens in two different headers.

Attacker Strategies. The probability of detecting a VAA covert channel depends on the strategy an attacker uses to mimic application traffic. The attacker can mimic a message according to two different strategies: the attacker creates messages by combining the list of elements (e.g., headers and header values) that were previously observed from the target application, and substitutes one header value with her secret data; or the attacker copies one of the previously observed messages from the target application entirely, and hides her secret data in one of the message headers. Due to a lack of space, we discuss the evaluation of the former, which is the same used by Chameleon. Since we assume an attacker cannot distinguish honey traffic from normal traffic, the attacker chooses the values to adapt at random. For this reason, *our evaluation assumes that the attacker mimics messages, or elements, uniformly at random.*

Adaptation per Header. Let us assume the attacker runs a collection phase for a period of time c, where it collects: (i) a header set $L = \{L_1, \ldots, L_j\}$ where L_i represents a list of headers; and a list of values V_{head} containing all the values associated with header *head* in the observed traffic. V_{head} also contains the observed honey tokens generated for header *head* during c, which we define as t_{head}. After choosing a list of headers L_i, the attacker generates an adapted message by randomly choosing a header value from V_{head} for each corresponding $head \in L_i$. Then, the probability p to detect an adapted malicious message with this strategy is $p = 1 - \prod_{head \in L_i}(1 - \frac{t_{head}}{|V_{head}|})$.

The probability of detecting an attacker after sending x messages, during blending phase, can be modeled as a binomial distribution, and it can be defined as $P(\text{detection within } x \text{ messages}) = 1 - (1-p)^x$.

Parameters. We now evaluate the effectiveness of honey traffic assuming the target application is a browser. First, we collected one hour of browsing activities using BurpSuite (e.g., streaming videos in the background and moderate web browsing) to estimate the number of requests an active browser may generate. We observed that roughly 2,200 requests were generated.

Consequently, within a collection timeout $c = 30\,\text{s}$, the number of requests a browser generates, which is also collected by the attacker, is 19. Therefore, we defined the number of application messages within c as $A_c = 19$. We choose a short collection timeout because malware may want to communicate closely to the benign application to avoid suspicion. Since we assume there are specific headers always present in application traffic, we define $|V_{head}| = A + t_{head}$, because the values observed from a single header contains all those generated by the normal application A and those generated by the honey client containing a token t_{head}. Finally, we evaluate our mitigation according to different amounts of honey messages generated during c. We define four cases, where the honey client generates 1, 2, 5, and 10 messages. The network overhead introduced by the honey client for these values is approximately 5%, 10%, 20% and 50%, respectively (e.g., for 19 browser messages in c, we introduce 1 honey message, which represent roughly the 5% of the browser traffic).

We want to remark that the detection probabilities are still influenced by the number of messages generated by the target application and the honey client, the collection timeout and how tokens are hidden. We evaluate HoneyTraffic with the parameters mentioned above to estimate the practicality of this method.

Results. The results show that HoneyTraffic can detect the attacker after few messages. The attacker can be detected with 80% probability after she sends 16 messages, costing only 1 message every 30 s. The increase of honey messages allows a faster detection. The results are shown in Fig. 3. Thus, it becomes problematic for an attacker to persist on the host, because she cannot perform many malicious operations before she is detected. Additionally, another important characteristic of honey traffic is that it generates almost no false positives, as long as tokens are unique in the traffic.

By multiplying the number of honey messages (within c) with the average size of an HTTP request, we can estimate how many kB/s our solution costs. Let us assume a pessimistic scenario, where the average size of a message is 1 kB. By using 0.03 kB/s (i.e., 5% overhead, 1 request in 30 s) of extra bandwidth, the attacker is detected with 90% probability after 23 messages. The overall daily cost in terms of bandwidth is 2.59 MB, for each browser we want to mimic in the network, and this is a worst-case scenario because we assume the browser is active 24 h. If we consider a 50% overhead, which is the pessimistic scenario in terms of bandwidth overhead, the bandwidth daily cost for a browser is 28.5 MB.

Table 4. Storage costs for the honey client, assuming each token is used only once (i.e., one-time tokens) and the setup phase is repeated once a week. We consider tokens to have an average size of 10 bytes.

Bandwidth overhead	Unique tokens	Storage size [kB]
5%	5760	403
10%	11520	806
20%	28800	2,016
50%	57600	4,032

The storage costs for the honey client are also relatively low. We evaluate a scenario where the setup phase waits one week to refresh the tokens, each token is used only once and is 10 bytes long. These are pessimistic settings since many tokens should be generated and hold in memory for a long time. For a bandwidth overhead of 50%, storing the tokens approximately costs 4,032 kB. A lower bandwidth of 5% has 403 kB of memory costs. Table 4 shows the costs in terms of storage of the honey traffic. In case the setup phase is executed more often, the memory costs decreases. For instance, a setup phase refreshing every day would cost 576 kB in case of a 50% bandwidth overhead.

This evaluation makes rough estimates about the performance of HoneyTraffic, and it shows that HoneyTraffic is a practical solution against specific VAA attackers strategies with negligible false positive, low bandwidth overhead and memory costs. As discussed in the section below, HoneyTraffic does not cover all the use cases of VAA covert channels, thus it cannot be considered a standalone solution. Note that this evaluation assumes the attacker adapts every time she sends a message. In case the attacker would send multiple messages with the same collected data, our evaluation still holds, but the X-axis of Fig. 3 should be interpreted as "adaptation attempts", instead of messages. Finally, note that this analysis assumes browser messages are evenly spread across different collection phases. We do not expect this to happen in practice, because the browser generates bursts of messages. Thus, some collection phases are more favorable to the attacker and some to the defender.

4.5 Evasion of Honey Traffic

Honey traffic cannot be proven secure, because we assume the attacker can run with high privileges, so, *in theory*, the malware can do anything on the compromised host. For instance, the attacker may be able to identify the process of honey client at system level and avoid mimicking packets that it generates. Nonetheless, we believe an attacker, especially if automated such as malware, faces severe difficulties in identifying the honey client, especially if properly hidden (e.g., by using rootkit techniques). The setup phase of the honey traffic is also a limitation, because the malware can get to know the list of tokens that will be used. However, also in this case, the malware must be aware of the presence

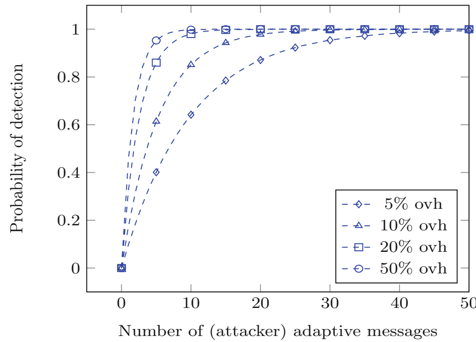


Fig. 3. Probability of VAA malware being detected by the number of messages it sends. The percentage values represent the overhead of honey messages, compared with the amount of browser traffic.

of the honey client on the system. Moreover, it should also be able to intercept and interpret the messages provided by the server.

Malware can evade honey traffic by generating its own traffic. However, by doing so the malware is not adapting, thus it would be detected by already existing techniques, as we discussed in Sect. 4. Alternatively, malware may try to corrupt, or even delete, the honey tokens by overwriting header values with valid strings, such that honey messages are never detected. This type of attacker is not represented by Chameleon, and thus it is not covered by HoneyTraffic. Nonetheless, it remains unclear whether an attacker substituting several header values with randomly generated valid strings can still be considered "adaptive". The generation of these strings may introduce distinctive patterns that could be identified by existing solutions. Due to these uncovered VAA strategies, HoneyTraffic cannot be considered a standalone solution to detect VAA covert channels. However, it can complement existing detection methods to make evasions more difficult for VAA attackers and provide overall better network protection.

5 Conclusions

In this paper, we presented Chameleon, a (to be released) toolchain to generate synthetic datasets containing adaptive covert channels—attacks that aim at mimicking the legitimate traffic generated by the victim while hiding secret information without being detected by a security monitor. Leveraging Chameleon, we showed that current detection approaches are not suitable for such advanced attacks. In fact, adaptive covert channels break their underlying assumption that malicious traffic presents distinctive patterns from benign traffic. We then proposed HoneyTraffic, a deception-based detection technique, which can complement existing detection mechanisms to make it harder, but not impossible, for attackers to evade detection using VAA covert channels.

References

1. Rossow, C., et al: Sandnet: network traffic analysis of malicious software. In: BADGERS, pp. 78–88. ACM (2011)
2. Stone-Gross, B., et al.: Your Botnet is my Botnet: analysis of a Botnet takeover. In: CCS, pp. 635–647. ACM (2009)
3. Continella, A., et al.: ShieldFS: a self-healing, ransomware-aware filesystem. In: ACSAC. ACM (2016)
4. Continella, A., Carminati, M., Polino, M., Lanzi, A., Zanero, S., Maggi, F.: Prometheus: analyzing webinject-based information stealers. J. Comput. Secur. **25**, 117–137 (2017)
5. Wendzel, S., Zander, S., Fechner, B., Herdin, C.: Pattern-based survey and categorization of network covert channel techniques. ACM CSUR **47**(3), 50 (2015)
6. Houmansadr, A., Brubaker, C., Shmatikov, V.: The parrot is dead: observing unobservable network communications. In: IEEE S&P, pp. 65–79 (2013)
7. Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T.: Protocol misidentification made easy with format-transforming encryption. In: CCS, pp. 61–72. ACM (2013)
8. Wright, C.V., Coull, S.E., Monrose, F.: Traffic morphing: an efficient defense against statistical traffic analysis. In: NDSS (2009)
9. Moghaddam, H.M., Li, B., Derakhshani, M., Goldberg, I.: SkypeMorph: protocol obfuscation for tor bridges. In: ACM CCS 2012, pp. 97–108 (2012)
10. Weinberg, Z., Wang, J., Yegneswaran, V., Briesemeister, L., Cheung, S., Wang, F., Boneh, D.: StegoTorus: a camouflage proxy for the tor anonymity system. In: ACM CCS 2012, pp. 109–120 (2012)
11. FAKEM RAT: Malware Disguised as Windows Messenger and Yahoo! Messenger. https://www.trendmicro.de/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fakem-rat.pdf
12. Bortolameotti, R. et al.: DECANTeR: detection of anomalous outbound HTTP traffic by passive application fingerprinting. In: ACSAC, pp. 373–386. ACM (2017)
13. Borders, K., Prakash, A.: Web tap: detecting covert web traffic. In: CCS, pp. 110–120. ACM (2004)
14. Schwenk, G., Rieck, K.: Adaptive detection of covert communication in http requests. In: EC2ND, pp. 25–32. IEEE (2011)
15. Fogla, P., Sharif, M.I., Perdisci, R., Kolesnikov, O.M., Lee, W.: Polymorphic blending attacks. In: USENIX Security 2006 (2006)
16. Fogla, P., Lee, W.: Evading network anomaly detection systems: formal reasoning and practical techniques. In: ACM CCS 2006, pp. 59–68 (2006)
17. Davis, J.J., Foo, E.: Automated feature engineering for HTTP tunnel detection. Comput. Secur. **59**, 166–185 (2016)
18. Casenove, M.: Exfiltrations using polymorphic blending techniques: analysis and countermeasures. In: Cyber Conflict: Architectures in Cyberspace (CyCon), pp. 217–230. IEEE (2015)
19. Yarochkin, F.V., Dai, S.-Y., Lin, C.-H., Huang, Y., Kuo, S.-Y.: Towards adaptive covert communication system. In: PRDC, pp. 153–159. IEEE (2008)
20. EMOTET Returns, Starts Spreading via Spam Botnet. https://blog.trendmicro.com/trendlabs-security-intelligence/emotet-returns-starts-spreading-via-spam-botnet/
21. PNFilter Malware Now Exploiting Endpoints, Not Just Routers. https://duo.com/decipher/vpnfilter-malware-now-exploiting-endpoints-not-just-routers

22. Zarras, A., Papadogiannakis, A., Gawlik, R., Holz, T.: Automated generation of models for fast and precise detection of http-based malware. In: PST, pp. 249–256. IEEE (2014)
23. Han, X., Kheir, N., Balzarotti, D.: Deception techniques in computer security: a research perspective. ACM Comput. Surv. **51**(4), 80:1–80:36 (2018)
24. Durumeric, Z., et al.: The security impact of https interception. In: NDSS (2017)
25. Sharafaldin, I., Lashkari, A.H., Ghorbani, A.A.: Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: ICISSP (2018)
26. Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: S&P, pp. 305–316. IEEE (2010)
27. Borders, K., Prakash, A.: Quantifying information leaks in outbound web traffic. In: IEEE S&P 2009, pp. 129–140 (2009)