The final version of this paper appears on Elsevier "Computer Networks" (COMNET), Vol.53, No.1, pp. 81-97, Jan. 2009.

Tunnel Hunter: Detecting Application-Layer Tunnels with Statistical Fingerprinting

M. Dusi, M. Crotti, F. Gringoli, L. Salgarelli

DEA, Università degli Studi di Brescia,via Branze, 38, 25123 Brescia, Italy

Abstract

Application-layer tunnels nowadays represent a significant security threat for any network protected by firewalls and Application Layer Gateways. The encapsulation of protocols subject to security policies such as peer-to-peer, e-mail, chat and others into protocols that are deemed as safe or necessary, such as HTTP, SSH or even DNS, can bypass any network-boundary security policy, even those based on stateful packet inspection.

In this paper we propose a statistical classification mechanism that could represent an important step towards new techniques for securing network boundaries. The mechanism, called Tunnel Hunter, relies on the statistical characterization at the IP-layer of the traffic that is allowed by a given security policy, such as HTTP or SSH. The statistical profiles of the allowed usages of those protocols can then be dynamically checked against traffic flows crossing the network boundaries, identifying with great accuracy when a flow is being used to tunnel another protocol. Results from experiments conducted on a live network suggest that the technique can be very effective, even when the application-layer protocol used as a tunnel is encrypted, such as in the case of SSH.

Key words: Traffic Identification, Encrypted Traffic, Network Security

1 Introduction

Firewalls and Application Level Gateways (ALG) have been used to secure network boundaries for many years. In their current incarnations, they are usually configured to protect local networks from at least two classes of attacks. First, they aim at controlling which sites local users connect to, and which application protocols are approved for use. This type of policy helps prevent damages that unsuspecting users might cause to their own network, for example downloading computer viruses by visiting rogue websites, or by

exposing to the Internet sensitive business information. Second, they (try to) limit attacks coming from the Internet, including the ones that are caused by worms and other network pests.

Security policies implemented by means of firewalls and ALGs start by defining which application-layer protocols are allowed, and which destination addresses can be contacted through such applications. The two type of devices are then setup to cooperatively enforce the policies: the firewall checks TCP ports and destination addresses, while the ALG verifies that the nature of the traffic crossing the network boundary is conforming to the policies, and that it is not malicious. For example, in the case of a policy allowing Internet browsing, the firewall might be configured to allow outgoing traffic to TCP port 80 if it is coming from an HTTP proxy. The proxy then checks that the peers are actually "speaking" HTTP. In addition, it verifies that the application-layer content of the connection conforms to the desired security policy, e.g., by denying connections to URLs containing particular strings or to sites whose name resolves to forbidden IP address.

However, several techniques have been designed in the last few years for disguising one protocol as another, with the specific objective to bypass the security policies at network boundaries. In these cases, not only such policies can become ineffective, but indeed could lead to a dangerous illusion of security. These techniques rely on tunneling one application-layer protocol into another one, i.e., they basically encapsulate the otherwise prohibited traffic inside the payload of allowed application protocols. Some tools even perform obfuscation of the original data, to make the tunneled connection not detectable by existing payload-based and pattern-matching classifiers employed by ALGs.

Tunneling techniques can be based on clear-text application protocols such as HTTP: in this case, a tunnel entry point within the LAN is configured to encapsulate all outgoing traffic into semantically valid HTTP requests that are then sent to a tunnel exit point, located outside the boundary of the LAN. This technique, in all its simplicity, can completely defeat any policy enforced by firewalls and ALGs [1].

Another tunneling technique is readily available with any Secure Shell (SSH) [2] implementation, that can be configured to protect any TCP traffic stream between an SSH client (typically the tunnel entry point) and an SSH server (the exit point) by means of cryptography. In this scenario, network administrators that intend to allow their users to open remote-terminal or secure-copy SSH sessions towards the outside world face a difficult choice. They can allow SSH to go through the boundary of their network, but in this case they lose any control over what the users may port-forward over SSH. For example, users might tunnel mail traffic through an SSH connection, potentially exposing their intranet to viruses and worms, since such mail would not be

sanitized by boundary devices. On the other hand, completely blocking SSH connections might not be possible, for example when the use of such protocol to protect remote-terminal sessions is essential to the job performed by the users.

In this paper we present a technique that, by characterizing *legitimate* uses of a given application-layer protocol, can enforce network-boundary security policies, helping firewalls block tunneled traffic. The key idea is that the information carried by packets at the network layer, such as packet-size and inter-arrival time between consecutive packets, are enough to infer the nature of the application protocol that generated those packets. By characterizing, or *fingerprinting*, the allowed protocols when used in their native form, we show that it is possible to detect with great accuracy when they are being used to tunnel other protocols, even when the tunneling mechanism uses encryption to protect the privacy of the traffic itself, such as in the case of SSH. The mechanism we present, called Tunnel Hunter, follows our previous work [3] and extends the results contained in [4,5].

The main research contributions of this paper are:

- the formalization of a statistical model, based on established pattern recognition approaches, that can be used to characterize application-layer protocols;
- the definition of a simple classification algorithm that can discriminate when an application protocol, characterized with the technique mentioned above, is being used to tunnel another protocol on top of it;
- the report on a series of experiments carried out on a real network that prove the effectiveness of the technique we propose here;
- the analysis of how adding knowledge to the classifier, even when not directly related to the classification target, can significantly improve the accuracy of the tunnel detector.

The rest of the paper is organized as follows. In Section 2 we report of related works, and briefly compare them to Tunnel Hunter. Section 3 is dedicated to an overview of the most common tunneling techniques used today to circumvent network-boundary security policies. In Sections 4 and 5 we introduce background information on pattern matching techniques, and use such information for the definition of the basic Tunnel Hunter mechanism. Section 6 reports experimental results on the detection of tunnels created on top of both HTTP and SSH. In Section 7 we further refine the basic mechanism, showing how increasing the knowledge of the classifier can improve the tunnel detection rates. Finally, Section 8 discusses potential attacks to our technique, while Section 9 concludes the paper.

2 Related work

Borders and Prakash proposed one of the first mechanisms to detect HTTP tunnels, called "Web tap" [6]. Their filtering mechanism was designed to detect spyware and backdoors over HTTP traffic. Although Web Tap's target is similar to ours, i.e., the detection of (malicious) traffic tunneled on HTTP, its mechanisms of operation are completely different. In fact, Web Tap relies on the analysis of features at the HTTP layer, such as HTTP transaction rates, transaction times, etc. Tunnel Hunter instead relies solely on the analysis of the transport layer: therefore it is able to detect tunnels built on top of other protocols, even when encryption is used, such as with SSH.

Recently Levine et al. [7] proposed a statistical technique that can breach the confidentiality of encrypted HTTP streams. Given a website, the authors collect the lengths of the packets composing each HTTP request: the mean value of each packet length traces the *size profile* of the site. In the same way, they derive the *time profile* from the inter-arrival times of the same packets. They then compare the profiles of several websites to recorded HTTPS traces: their cross-correlation is then used to expose the similarity of the traces to each given profile, making it possible to assess the destination address of the corresponding flow, even if it is encrypted. Liberatore et al. in [8] follow a similar approach, evaluating the technique with a larger data set of web-profiles. They compare two different methods to assign a given trace to a gathered profile, i.e., the Jaccard and the näive Bayes similarity metrics.

In the work of Bonfiglio et al. [9] two techniques to identify Skype voice calls are described. The authors combine the use of statistical properties of message content and näive Bayesian techniques to reveal the voice streams from an aggregate of UDP traffic. Since UDP Skype packets are not completely obfuscated and contain some statistical bias, it is possible to state if the content of an UDP stream is completey random (i.e., the bytes are uniformly distribuited) or not (it belongs to a Skype UDP session). A subsequent check with a näive Bayes classifier, trained to detect VoIP streams can identify Skype calls with a high degree of confidence. Another work that deals with privacy in encrypted data streams is the one by Wright et al. [10], where it is shown that encrypted IPSec tunnels which carry only a single application protocol leak enough information about the flows in the tunnel to allow to precisely assess their number. The authors also introduce techniques to track flows without de-multiplexing or reassembling the TCP connections in the aggregate. The main implication of these works is that encryption is not enough to protect privacy when statistical mechanisms are used to analyze traffic flows. However, none of these mechanisms are targeted at blocking tunneled traffic according to pre-set security policies.

Moving away from encrypted traffic, the analysis of plain-text Internet traffic to detect application-layer protocols has a long history. The use of Deep Payload Inspection (DPI) techniques is today very popular for the classification of network traffic and is implemented in many Network Intrusion Detection Systems such as BRO and Snort [11,12]. However, the ability to build pattern-matching regular expressions that can effectively classify tunneled traffic has yet to be demonstrated, and it becomes completely ineffective anyway when employed on encrypted tunnels.

A novel kind of approach that could overcome these impairments is based on the analysis of traffic from a statistical point of view as opposed to the deterministic techniques used in DPI-based mechanisms. Starting with the seminal studies of Paxson such as [13], researchers have proposed several algorithms based on traditional classification techniques such as hierarchical clusterization [14,15], Nearest Neighbor and Linear Discriminant Analysis [16], and Bayesian Learning Machine [17]. The list of statistical approaches to traffic classification is growing quite long, but to the best of our knowledge, none of these techniques has yet been tested on tunneled traffic with the purpose of strengthening the application of network-boundary policies.

In our own previous work [4] we used a statistical traffic classification technique that we developed in [3] to detect when a HTTP connection is carrying another protocol encoded inside the application payload. In [3] the technique was exploited to recognize different kind of application protocols, such as POP3, SMTP and HTTP. In this paper we extend our previous works in several ways. First, we provide a framework to our technique, by formalizing it as pattern-recognition problem and by exploiting a rejection schema in our classification mechanism, as described in [18–20]. Second, we expand the tests to verify the effectiveness of our technique in detecting HTTP tunnels: here we deal with a more sophisticated HTTP tunneling technique and more types of tunneled traffic, such as peer-to-peer (P2P). Third, we enhance the technique to deal with encrypted SSH tunnels, extending it to detect the different behaviors (i.e., usages) of a single application protocol, even if encrypted. Furthermore, in Section 7 of this paper we shed some light on the effect of adding more knowledge to classes that are not directly related to the target of the classifier, proving that the presence of additional classes can indeed improve the overall accuracy of the tunnel detector.

3 An overview of tunneling techniques

The goal of tunnel mechanisms is to disguise a given application protocol as another one. In order to elude the ALGs, the process produces a stream of packets that, at the application-layer, are exactly conforming to the allowed

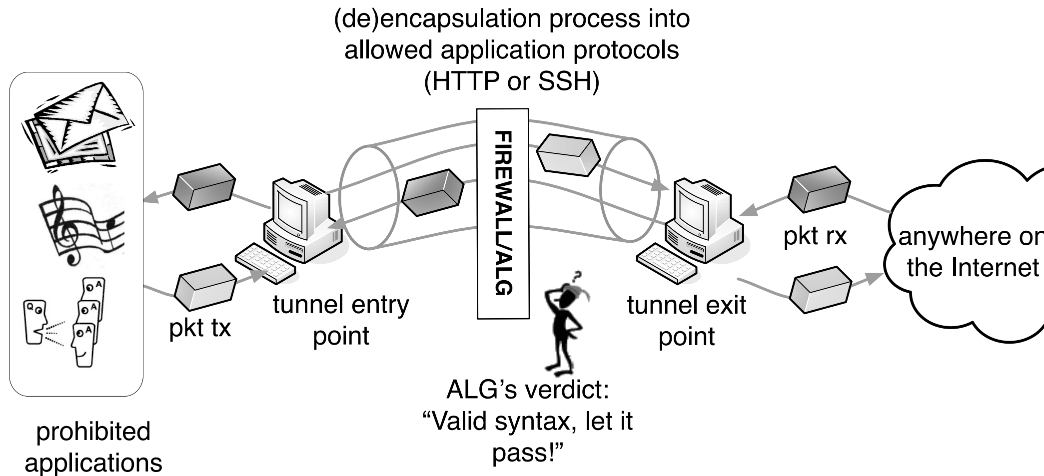

Fig. 1. How tunnels work: high-level scheme.

protocol(s). Regardless of the protocol used as tunnel, tunneling mechanisms basically operate with a client-server model: a client host inside the protected network connects to an outside server using an application protocol that is allowed by the network-boundary security policies. Each endpoint then provides the (de)encapsulation of the tunneled protocol and forwards the original data to the actual server or client involved in the communication. Full control of the tunnel endpoints is required, but this is usually not a problem: common setups involve a laptop inside the office and a home computer connected to the Internet through a DSL line, or a computer on any other network connected to the Internet. A high-level scheme of a generic tunneling technique is depicted in Figure 1.

At least three protocols can be used to tunnel Internet traffic at the application-layer: DNS, HTTP and SSH.

3.1 DNS tunnels

Application-layer tunnels can be built on top of DNS by simply exploiting the way regular DNS requests for a given domain are forwarded to its authoritative servers. The entry point breaks up the IP packets to be tunneled inside requests for the domain whose authoritative server is the tunnel exit point: other DNS servers (resolvers) laying between the protected network and the Internet will forward these requests to the tunnel exit point, which in turn reassembles the original packets and sends them to their actual destinations. DNS responses will then be used to deliver packets on the opposite direction to the tunnel entry point. Since the DNS protocol is rarely blocked on the Internet, this technique can be very powerful: a good implementation is available at [21]. Due to the mechanism's complexity, however, DNS tunnels can rarely achieve throughputs higher than a few kb/s, and are therefore seldom

used. In this paper we consider more efficient solutions based on HTTP and SSH connections on top of TCP.

3.2 HTTP tunnels

Network administrators normally let HTTP traffic pass their network boundaries, even though usually filtering it through an application-layer proxy and a firewall. However, both legitimate users and malicious network software (viruses and malware) can violate these policies by using the HTTP protocol to tunnel any other application through the network-boundary, as depicted in Figure 1. The packets of the tunneled flow are encoded so that they can be incorporated in one or more regular, semantically valid HTTP sessions. An analysis of the payloads of each session, even if performed by means of pattern matching, could not reveal any difference between the tunneled traffic and a genuine HTTP flow generated by actual web-browsers and web-servers.

One of the HTTP tunneling techniques most commonly used these days is implemented by *httptunnel* [22]. This tool provides a pair of daemons running at the tunnel end-points. At the entrance side, *httptunnel client* waits for incoming TCP connections on a configured port. When a connection is established, it initiates a HTTP session towards the *httptunnel server* running at the tunnel exit point. A typical scenario could be the following: let us assume that the security policies of an enterprise network allow web-browsing traffic but block all outgoing TCP connections to port 110, so that users can only retrieve e-mail messages from trusted, internal POP3 servers. A user, wanting to access its private e-mail account outside the company, could easily violate the policy by running a HTTP tunnel. To this end, he or she would configure *httptunnel server* on their computer at home listening on port 80 and *httptunnel client* on their company laptop to forward incoming connections to port 110 over the tunnel: the commands required to implement such tunnels would be:

```
officepc:~> httptunnel -cli \
                110:homepc.org:80:private.email.com:110
homepc:~> httptunnel -svr 80
```

The first line, executed on the office laptop, sets up the tunnel entry-point: every incoming connection to port 110 will be forwarded to the home computer that will then connect to the private POP3 server. The second line, instead, sets up the computer at home to act as a HTTP server.

More in details, whenever a new tunnel is established, the two daemons talk by means of HTTP GET messages, which are periodically sent by the tunnel entry to (i) poll the exit and check if new data is available from the remote

side and (ii) send new data that was generated locally.

In our previous work [4] we tested the open source package *htc/hts* [23], where the tunnel is established by opening two connections between the tunnel end-points. A connection is used to deliver outgoing traffic encapsulated inside HTTP POST messages, while the other connection fetches traffic from the remote side through HTTP GET requests. In this paper, we aim our tests at tunnels generated by *httptunnel*, which are tougher to detect for several reasons. First of all, *htc/hts* relies on two simultaneous HTTP connections, as opposed to the single one used by *httptunnel*. In this sense, the classifier can have an easier time spotting *htc/hts* tunnels by coordinating the analysis on the two HTTP connections. Second, *htc/hts* relies on the HTTP POST message, which could be detected (and blocked) by an ALG at the first signs of tunneling activity. This kind of technique would be ineffective with *httptunnel*, since it relies only on "safe HTTP methods" (i.e., "GET"), which are mandatory in any HTTP implementations, according to [24].

3.3 SSH tunnels

The SSH protocol runs on top of TCP and it is designed to provide data confidentiality and integrity between two hosts over an insecure network. This protocol is typically used for command execution through a secure shell and secure file copy between peers. It also supports tunneling of arbitrary TCP connections, also known as *port forwarding*. Tunneling over SSH cryptographically protects otherwise insecure protocols, increasing data and system security. However, as a result of data encryption, any network-boundary security policy that relies on a DPI technique is totally neutralized. While in the case of HTTP tunnels there might be advanced ALGs which, based on deep-packet-inspection, could conceivably ascertain the true nature of tunneled flows, in the case of SSH the research of well-known payload signatures is completely useless, making tunnels built on SSH very powerful. Network-boundary security policies that allow the use of SSH for remote command execution or secure file copy have no other choice than to allow, with today's ALGs, that any protocol can be tunneled in and out of the intranet by means of SSH tunnels.

SSH is designed following a client-server model: the server is usually implemented with a daemon running in background and accepting connections to port 22, which clients connect to. A typical setup to establish the same type of tunnel as the one described in Section 3.2 with the OpenSSH package [25] is the following:

```
officepc:~> ssh -L110:private.email.com:110 user@homepc.org
```

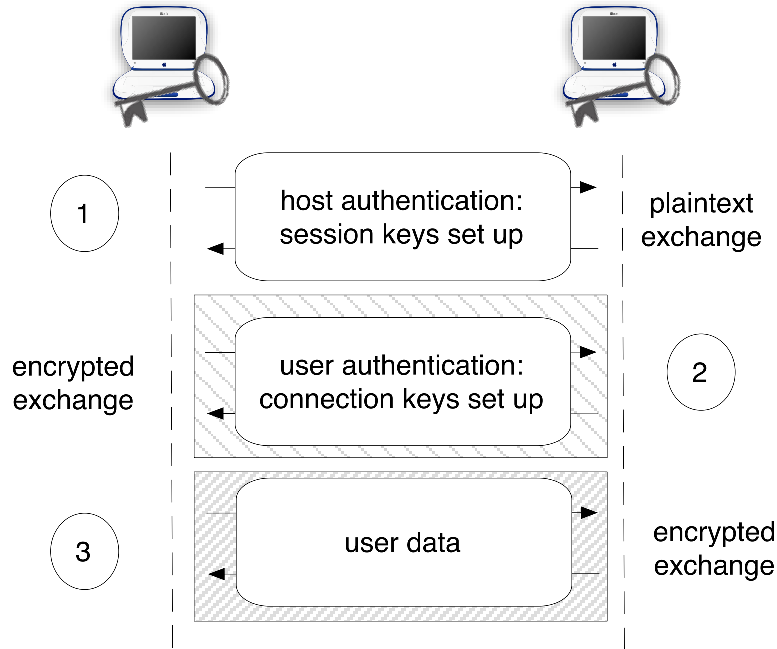

Fig. 2. Authentication stage in SSH.

This example establishes a secure connection between `officepc` and `homepc` where we suppose that an SSH server has been properly installed. The SSH client process on the first host will accept incoming TCP connections to port 110, securely tunneling them to the SSH server that will in turn connect to the private email provider on behalf of the actual source of the communication.

Data encryption is ensured by the SSH protocol that, before any traffic can be exchanged, requires the peers to negotiate cryptographic credentials [26,27]. Irrespective of the nature of data being exchanged with SSH, be it remote-command execution, secure copy or port forwarding, the two peers must go through a two-phase authentication stage. This is a significant difference from clear-text HTTP tunnels, which require no authentication. Since the authentication procedures of SSH impact the way Tunnel Hunter is implemented, we need to go into more details on how they work.

3.3.1 The SSH authentication process

An SSH session involves two different authentication phases before client and server can start exchanging data: (i) host authentication and (ii) user authentication. Figure 2 outlines the SSH authentication process.

Host authentication, as reported in [27], provides strong encryption, cryptographic host authentication, and integrity protection. The key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated. It is expected

that in most environments only two round-trips are needed for full key exchange, server authentication, service request, and acceptance notification of service request: in the worst case, an extra round-trip could be required. This authentication phase is transmitted un-encrypted and ends with an `SSH_MSG_NEWKEYS` message. All messages sent after this must use the negotiated keys and algorithms, and are privacy and integrity protected.

User authentication, as reported in [26], is intended to be run over the SSH transport layer protocol, i.e., the encrypted channel derived by the host authentication phase. Public-key is the only mandatory authentication method, although passwords are also accepted. Successful password authentication in SSH requires a single round-trip: the client transmits the password to the server, that replies with an ACK or a NACK. In the last case, the client has other chances to re-send the correct password. The public-key method can either require one or two round-trips: the specifications have defined an optional initial exchange where the client could send information on its public key to the server before sending a signature with its own private key.

Packets exchanged during the entire SSH authentication process are not useful to a classifier to detect the type of session that the user is opening, i.e., whether the session is tunneling other protocols or is being used for remote command execution or secure file copy. The classifier can easily discard all the packets exchanged up to the client's `SSH_MSG_NEWKEYS` message and the server response, since they are sent in the clear. It is more difficult to detect when user authentication ends, and actual data starts being exchanged, because the second authentication stage is encrypted. In this paper we solve this issue assuming that network administrators are required to choose and allow only one kind of SSH user authentication method to implement Tunnel Hunter on their networks.

We will discuss this and other aspects related to the way SSH configuration parameters affect our mechanism in Section 6.2.

4 Background on statistical pattern recognition

Tunnel Hunter is based on mechanisms which are directly derived from the branch of mathematics known as statistical pattern recognition. To provide a framework for our technique, here we introduce some of the basics of pattern recognition theory.

4.1 Terminology

There are three main definitions at the basis of statistical pattern recognition: pattern, feature and class [28]. The *pattern* is an r-dimensional data vector $\vec{x} = (x_1, \ldots, x_r)$ of measurements, whose components x_i measure the features of an object. A *feature* represents the variable specified by the investigator and holds a key role in classification. The concept of *class* is used in discrimination: assuming a set of C classes, named $\omega_1, \ldots, \omega_C$, each pattern $\vec{x}$ will be associated with a variable that denotes its class membership. If the classes are gathered from an *a priori training set*, that is, the information about how to group the data into classes is known and it is not inferred directly from the data itself, the approach is called supervised, otherwise it is called unsupervised.

Given a set of data and a set of features, the goal of a pattern recognition technique is to represent each element as a pattern, and to assign the pattern to the class that best describes it, according to chosen criteria.

4.2 Class description

The class description depends on the chosen approach, supervised or unsupervised. In a un-supervised (or clustering) approach, the data are seen as a whole and the goal is to infer some information about the data by means of blind techniques: the patterns are analyzed to form classes without any a-priori knowledge about the data. In a supervised approach the data are previously divided into different groups: the patterns related to the data belonging to the same group are used to define the model of each class. Since in this paper we will only consider supervised approaches, we will not take into account the theory behind unsupervised methods.

Once groups - classes - have been identified, one can collect observations coming from each class ω_i and create a training set $\mathrm{T}_s(\omega_i)$. The target at this stage of the analysis is to build models that can be then used to classify new observations as they come: a thorough inspection of T_s can lead to an analytical model describing the corresponding class. Once knowledge has been acquired, one has to determine a decision function f that takes the observed data $\vec{x}$ as input and outputs a prediction of the class that generated it, $\hat{\omega}(\vec{x}) = f(\vec{x})$, minimizing the error of incorrect decision. The most commonly used classification functions are based on *Maximum Likelihood* or *Bayesian Decision* estimators [29]. In general, these methods exploit the conditional densities $p(\vec{x}|\omega_i)$ to design the optimal decision rule f.

It happens frequently, however, that it is not possible to develop a complete

and accurate analytical model: in this case one has to adopt a statistical learning approach to develop estimators based only on a collection of training samples [30,31]. When obtaining training data is a simple task, an option is to estimate the joint probability distribution of the observed quantities, and then use this estimate to derive a decision rule.

In this paper we focus on density estimation since, when dealing with network traffic, the collection of proper amount of training data (non-tunneling traffic flows) is a feasible task. Once the data are converted into their equivalent pattern representations, a histogram method can be adopted to provide a non-parametric density estimation of the class. This method partitions the r-dimensional space of class ω_i into a number of equally-sized cells and estimates the density at a point $\vec{x}$ as follows:

$$\hat{p}(\vec{x}|\omega_i) = \frac{n_j}{\sum_{j \in N} n_j dV},$$

where n_j is the number of samples in the cell of volume dV that straddles the point $\vec{x}$, N is the number of cells and dV is the volume of each cell. If the pattern $\vec{x}$ is r-dimensional, the approach requires N^r cells to store each $\hat{p}$.

When the variables are independent, the class conditional probabilities can be derived as follows:

$$\hat{p}(\vec{x}|\omega_i) = \prod_{k=1}^{r} \hat{p}(x_k|\omega_i),$$

where $\hat{p}(x_k|\omega_i)$ is the individual (one-dimensional) density of the components of $\vec{x}$ conditioned to class ω_i. In this case the complexity of the problem decreases dramatically since there is no need to evaluate (and store) densities in a r vectorial space: each of the r marginal density estimates takes at most N cells. As in the analytical case, once the training phase has been carried out for each of the target classes, one can use a Maximum Likelihood Estimator to assign a new observation to the class that maximizes the likelihood function:

$$\hat{\omega}\left(\vec{x}\right) = f\left(\vec{x}\right) = \arg\max_{\omega_j} \hat{p}(\vec{x}|\omega_j),$$

or set up a Näive Bayes classification algorithm [32].

4.3 Stages in a pattern recognition problem

In its general form, a pattern recognition problem involves the following high-level steps:

Data collection: making measurements on appropriate variables and recording details of the data collection procedure; this step includes ascertaining the "ground truth" about the data for supervised approaches.

Feature selection or feature extraction: selecting variables from the collected data that can be useful for the classification task.

Definition of patterns and classes: describing data as a pattern of features and gathering one or more classes by means of (un-)supervised approach.

Definition and application of the discrimination procedure: design of a classifier that can operate on the class descriptions and emit a classification verdict as a new pattern comes.

Assessment and interpretation of results.

This often results in an iterative process: the analysis of the results may shed new light on the problem, requiring further data collection, feature extraction, measurements and so on.

5 Tunnel Hunter

Tunnel Hunter aims at detecting tunneling activities over the HTTP and SSH application protocols. We define "*tunneling activity*" (i) any HTTP flow which is used to carry a stream of packets generated by application-layers other than HTTP, and (ii) any SSH flow which is used to carry a stream of packets generated by anything other than interactive remote shells or secure file copying.

The detection mechanism is based on a simple principle: since characterizing precisely all possible non-legitimate activities (i.e., tunneling flows) is practically impossible, Tunnel Hunter focuses on *building an accurate description of legitimate traffic* (the target class), trying then to detect which flows to block (the outlier class) by comparing their behavior to the target class. Dealing with a well-defined target class and an ill-defined outlier class is a common situation in pattern recognition [19]. As we will see in the following, Tunnel Hunter builds on known pattern recognition techniques to achieve its objectives.

This section is divided into two parts. In 5.1 we describe how network traffic is processed to (i) transform TCP flows into patterns and (ii) create statistical fingerprints of a specific usage scenarios of an application protocol. In 5.2 we introduce the actual tunnel hunting algorithm, i.e., a statistical classifier trained on traffic generated by just the target class. Concepts introduced in these sections will be reused throughout Section 7 when we will deal with multi-class discrimination.

5.1 Building patterns and classes

5.1.1 Measurement of features for legitimate flows

Our technique models a legitimate (i.e., non-tunneling) behavior of an application protocol by extracting basic statistical features from the TCP session that carries it. The features are gathered directly at the IP-level and the pattern is derived straight from the flows composing the TCP session. We define "flow" as the unidirectional stream of packets exchanged throughout a TCP session: F_{client} is the flow that goes from the initiator to the responder, while F_{server} is the one in the opposite direction.

The features we choose to represent flows are the packet size s and the inter-arrival time Δt between two consecutive packets. The rationale behind this choice lies in the observation that at least during the beginning stage of each TCP connection, the statistics related to s and Δt for each packet of the flow depend mostly on the application-layer state machine that is generating the flow. This proves to be true, for example, for the authentication stage in a POP3 retrieval, for the SMTP helo-sender-receiver agreements, the HTTP data request, and so on. This assumption is also supported by other papers in the literature (see Section 2), and by our own previous work on statistical traffic classification [3], where we showed the effectiveness of inferring the nature of application protocols starting from s, Δt and packet arrival order.

Note that Tunnel Hunter does not to consider packets that do not carry TCP payload. In fact, empty TCP packets do not add any additional information useful for classification, since they only carry the "transport-layer signaling" required to setup and maintain a TCP session. In other words, the statistical behavior of TCP packets that do not carry any payload does not depend on the behavior of the application, but is a function of the TCP/IP stack of the end nodes, as well as of the status of the network, and is therefore of little use in the characterization of application-layer protocols.

Starting from the definition introduced in Section 4 we choose the following pattern to represent a TCP flow:

$$\vec{x} = \begin{pmatrix} s_1 & \dots & s_r \\ \log_{10}[\Delta t_1] & \dots & \log_{10}[\Delta t_r] \end{pmatrix}. \tag{5.1}$$

Here r represents the number of packets that are useful in our context, i.e., the ones that carry application layer payloads, indexed in ascending order as they are seen by the capture device. A TCP session is then represented by the union of the pair of patterns $\vec{x}_{F_{client}}$ and $\vec{x}_{F_{server}}$.

Variable s is discrete and is generally limited by the mix of data-link layer technologies on the path between the connected peers. Variable Δt, instead, exhibits substantial variations: it depends, in fact, on the distance of the peers, networking architecture, network congestion and many other factors. For this reason we adopt a logarithmic transformation and truncate the range of possible values to a limited interval: the resulting outcome is then quantized and the observed variable $\log_{10}[\Delta t]$ is discrete. In the following we will denote the cardinality of the domain of observable values for the two variables respectively as $N(s)$ and $N(\Delta t)$. We will go into details in a later section when dealing with numerical results.

5.1.2 Class model: the concept of protocol fingerprint

In order to build a classifier capable of distinguishing different usages of a given application protocol, we have to provide it with some knowledge. At first glance, one might think that if a protocol can be used for N different purposes, the classifier should be trained on data coming from each usage scenario, properly divided and organized in N classes. Although this approach could lead to very accurate classification, it is not always feasible, especially in the context of detection of application layer tunnels. For example, in the case of HTTP, one should gather knowledge about all possible usages of this protocol before being able to state when the protocol is used to transport web pages or to carry other application protocols on top of it. The question here is about how many classes one has to consider: for example, while the class of legitimate HTTP flows (i.e., related to web browsing activities) might be relatively simple to characterize, the precise definition of one class for each of the possible applications that can be tunneled on top of HTTP is practically impossible.

In this paper we report how we have successfully sorted out this issue, following two approaches with increasing complexity. One approach requires to train the classifier only with flows from a single target class: in the case of HTTP, the training is carried out on legitimate web browsing traffic, while in the case of SSH the target class is characterized by remote shell and secure file copying activities. We call this method *one class classifier* [33,30].

The second technique, instead, adds to the *well-defined* target class a subset of its *ill-defined* complement. To this end, new classes composed of outlier flows are added to the analysis: although they do not completely define the behavior of all the flows that are not generated by the target class, their inclusion adds knowledge to the algorithm and hence improve the classifier's abilities in the identification of target traffic. We call this method *multi-class classifier* [19,18]. We will underline pros of both approaches in the following sections.

In the rest of the paper we refer to C as the collection of all classes that represent the possible usages (either legitimate or not) of a given application protocol *App*. For example, in the case of SSH, C will be the union of the classes that represent remote command execution SSH flows, secure remote copy, tunneling of P2P on top of SSH, etc. We will refer to ω_t as the target class, contained in C, for a particular protocol. For example, in the case of the one-class algorithm applied to SSH, ω_t will be the class that represents the union of the two non-tunneling usages of SSH, i.e., remote command execution and secure copy.

To gather the statistical description of ω_t we start by collecting a set of flows generated by *App* within the usage context defined by the target class. During this step it is important to verify that the captured flows indeed belong to the target class, i.e., that they are not the result of tunneling activities. This can be achieved either by a thorough payload-based analysis or by artificially generating the traffic. In both cases, the collected flows compose the *training set* T_s of class ω_t. The set is then converted into a pattern representation, according to the procedure outlined in Section 5.1.1, and the histogram method is used to produce a non-parametric density estimation of ω_t.

Considering the variables that compose the pattern in Equation 5.1 correlated would lead to an intractable problem, since the estimation of the class-conditional density would require the management of a multi-dimensional domain counting as much as $\{N(s) \cdot N(\Delta t)\}^r$ elements. Therefore, we decided to characterize separately the behavior of each of the r column vectors inside pattern $\vec{x}$. We remark that this choice was made not because of hypotheses on the statistical independence of the consecutive packets that compose the observed flows: we opted for it because we needed a simple classification algorithm that could be easily implemented on real networks, and represent a lower bound in the accuracy of the target class description. As we will see, this assumption not only greatly reduces the mechanism complexity, but also leads to excellent results, thus making this choice sensible and supported by experimental results. The evaluation of the impact of the class-conditional densities on the classification performance is left as future work. In this context we imagine that the evaluation of the class-conditional densities should lead to a more accurate model of the class.

In order to be able to stop the classification process at any stage while examining a flow, we introduce an additional parameter L, which indicates the maximum number of column elements of the pattern in Equation 5.1 that can be taken into consideration when classifying a flow. Therefore, the output of the phase of pattern definition are L marginal density estimations (DE), each one defined in a $N(s) \cdot N(\Delta t)$ domain.

To sort out problems concerning possible sparseness of the resulting DE ma-

trices, a filtering step is carried out. It can happen, in fact, that densely populated matrix regions contain areas with zero-valued elements; at the same time, densely populated regions can abruptly fall to zero without smoothly decaying boundaries. These issues may mislead the description of the target class ω_t or, even worse, compromise its robustness to "noise" factors such as network congestion: once a DE is ready, a variation of the round trip time could introduce noise in the measurements of Δt and one of the packets of a new flow of the same kind could fall in a zero-valued cell of the matrix even if the surrounding cells are populated with non zero values. To counter these issues, following the well know kernel method (or Parzen method), we filter the DE matrices using either a Gaussian or a hyperbolic secant kernel. Finally, we normalize the DE matrices again so that they still sum up to 1.

Like a TCP session, that is represented by a pair of patterns $\vec{x}$, one per flow direction, the statistical description of the target class will be made of a pair of *fingerprints* - the computed DEs: one built on F_{client} flows, the other on F_{server}. These fingerprints capture a precise behavior of the considered application protocol, the one that falls inside ω_t, i.e., HTTP web-browsing, the SSH remote administration or Secure Copy (SCP). In the rest of the paper we will use symbol $\vec{M}$ to indicate either F_{client} or F_{server} fingerprint, specifying direction when needed.

5.2 One-class tunnel detection algorithm

We introduce here the decision function that we will adopt for tunnel detection in the one-class algorithm context. We will reuse the theory reported below also in Section 7 in the multi-class context. In the last part of this section, we show how to tune the basic algorithm.

5.2.1 Algorithm definition: the decision function

Our algorithm can be targeted to discover or "hunt" a particular usage scenario of a given application protocol: in this context it could be used to distinguish tunneling sessions from an aggregate of HTTP traffic or to discover when a encrypted SSH flow carries another TCP session on top of it. We start focusing on an implementation that simply relies on the characterization of the target class $\omega_t \subset C$ to discover TCP sessions generated by application protocol *App* that fall in subset $\omega_r \equiv C \setminus \omega_t$. By construction, ω_t represents the "legitimate" use of *App* or *acceptance region*, w_r is instead the *rejected region*, complementary to ω_t.

The algorithm takes an unknown flow generated by *App* and computes a metric using the information held by the fingerprint $\vec{M}$ describing ω_t. This metric

will then be used as input by the decision function f. As discussed in the previous section, fingerprint $\vec{M}$ models the behavior of each of the first L packets in the flow without taking into account any possible influence between consecutive packets. Every packet will hence supply a small contribution to the metric: pieces will be then combined independently. We expect that a more sophisticated algorithm designed to consider class-conditional densities could only lead to better classification results.

Given an unknown flow F, the algorithm maps its pattern representation $\vec{x}$ onto the fingerprint and returns an index of (dis-)similarity. We call this index *anomaly score* and define it as follows:

$$S(\vec{x}|\omega_t) = \left| \frac{\log_{10} \prod_{i=1}^{\min(r,L)} p(x_i|\omega_t)}{\min(r, L)} \right|. \tag{5.2}$$

Here $p(x_i|\omega_t)$ represents the marginal probability that the i-th element of the flow pattern belongs to class ω_t. By construction we can estimate this probability by looking at the $i-\text{th}$ slice inside $\vec{M}$, that means

$$p\left(x_i|\omega_t\right) \simeq M_i\left(s_i, \Delta t_i\right).$$

The closer the anomaly score of F is to zero, the more the flow fits the behavior of ω_t. On the contrary, a higher anomaly score value stands for a growing probability that the flow has been generated by other classes. It is worth noting that to avoid numerical inconsistencies in Equation 5.2, we replace every null valued $p(x_i|\omega_t)$ with a value near to the floating point precision, i.e. 10^{-300}. This value lets us deal with logarithms without affecting the accuracy of our classifier: actually, the contribution given by such a value to the anomaly score is big enough to unequivocally classify the flow.

It is noticeable that the anomaly score alone is not sufficient to classify a flow: this follows from the fact that we are only considering the acceptance region given by the target class ω_t inside the algorithm. If we had prepared the fingerprint of the rejected region too, we could calculate another anomaly score, this time conditioned to ω_r and decide for one of the two regions through a Maximum Likelihood algorithm comparing the pair of computed metric.

Since we still do not want to introduce the complexity of gathering a statistical description for the rejected region, we follow another strategy: we simply separate the acceptance region from the rejected one by introducing an acceptance threshold T_{acc} in the classification process. By construction, a low anomaly score stands for a high probability that the flow belongs to the class. Therefore, we must accurately calibrate T_{acc} so that it is higher than the anomaly scores of flows coming from the target class ω_t, while being low enough

so that non-legitimate flows would be assigned to the ω_r class. The process is summarized by the following expression, that define Tunnel Hunter's decision function f for the one-class case as:

$$\hat{\omega}\left(\vec{x}\left(F\right)\right) = \begin{cases} \omega_t \text{ if } S(\vec{x}|\omega_t) < T_{acc}, \\ \omega_r \text{ otherwise,} \end{cases}$$

where $S\left(\vec{x}|\omega_t\right)$ is the anomaly score of the flow F conditioned to the target class. In this context we call false positives not-legitimate sessions that our algorithm incorrectly assigns to ω_t. False negatives, instead, occur when our algorithm rejects sessions generated by the target class.

The precise calibration of the threshold is of primary importance for the correct behavior of Tunnel Hunter. We defined the following simple procedure to calibrate T_{acc}. We computed the anomaly score for a collection of selected flows grouped inside a new training set T''_s: these flows belong to the usage class described by ω_t and are not included in T_s so that $\mathrm{T}_s \cap \mathrm{T}''_s \equiv \emptyset$. We chose for T_{acc} the lowest value that allowed exactly 99% of the flows inside T''_s to enter ω_t. The rationale behind this procedure is that since flows from T_s and T''_s have the same statistical behavior, we expect that the classification of new unknown traffic should produce roughly similar results.

Note that, as for many other quantities in this paper, we need to differentiate the threshold depending on the direction of the flow. Therefore we have two thresholds for each fingerprint, one built on F_{client} flows and another one built on F_{server}. Again, we will specify the direction when needed.

The training phase has to be based on a set of flows whose nature can established with certainty. For example, we need recording of legitimate, non-tunneling HTTP sessions to characterize the ω_t class for HTTP traffic. In other words, we need to verify the ground truth as to which application really generated each flow we use during the training phase. We will see in the following section how we dealt with this issue in practice.

5.2.2 Tuning the one-class algorithm

Once the training set has been collected, it needs to be split into two different subsets: a training set T_s to build the fingerprint for the legitimate class ω_t and another T''_s to calibrate the free parameters of the system, which are three: the threshold, the type (and parameters) of kernel function, and the packet where to start the classification process.

The threshold can be calibrated by following the procedure outlined in the

previous section: T_{acc} is set to the value that allows 99% of flows in T''_s to fall into the ω_t class.

As for the kernel function used to filter the DE matrices (refer to Section 5.1.2), several types of kernels are commonly used in the literature. In this paper we focus on two of the simplest ones, and we compare the effects that a Gaussian $G(\vec{u})$ and a hyperbolic secant $\mathrm{Sech}(\vec{u})$ bi-dimensional kernels have on our density estimations:

$$G(\vec{u}) = \frac{1}{2\pi h^2} e^{-\frac{||\vec{u}||^2}{2h^2}} \qquad \mathrm{Sech}(\vec{u}) = \mathrm{sech}\frac{||\vec{u}||}{h},$$

where h is the bandwidth, or smoothing parameter, of the kernel. The robustness against "noise" factors essentially depends on the bandwidth of kernel function: intuitively, the higher the bandwidth, the lower the probability that the flow will fall in a zero-valued cell of the fingerprint. On the other hand, the higher the bandwidth, the lower will be the capacity of the algorithm in discovering if flows are non-legitimate.

The analysis of the impact of different kernel functions on the performance of the classification algorithm improves our previous works [4,5], where the role of the Gaussian kernel was not discussed. Here, instead, we thoroughly investigated the effect of the filtering stage on the fingerprints that describe legitimate activities. Since the protocol fingerprint can only be gathered starting from a limited data set, we proved indeed that the kernel function has a substantial impact on the ability of the fingerprint to provide a precise but compact description of the training data.

Another free parameter that can affect the precision of the classification algorithm is the choice of the first packet to be considered for analysis. Until here we implicitly settled on starting from the first packet after the three-way-handshake as can be noticed in the pattern described by Equation 5.1 and the definition of the anomaly score given in Equation 5.2. In the case of SSH tunnels, this choice could be counterproductive: as depicted in Section 3.3, the SSH protocol provides an initial, two-phase authentication stage that is independent on the use of the connection – remote command execution, secure copy or port forwarding. To take into account this fact, we introduce the concept of "shifted window": in this way we can explicitly specify the pairs $(s, \Delta t)$ that will be actually considered by the classification algorithm to compute the anomaly score. The intent is to have the anomaly score calculated only on the packets that represent actual data exchange, excluding the ones that carry authentication information. Therefore, Equation 5.2 has to be rewritten in a

more general form:

$$S(\vec{x}|\omega_t) = \left| \frac{\log_{10} \prod_{i=n_0}^{n} p(x_i|\omega_t)}{(n - n_0 + 1)} \right|, \quad (5.3)$$

where n_0 and n are the indexes of the first and the last considered pairs respectively. For instance, in the case of SSH traffic, n_0 should point to the first packet that carries actual user traffic in each SSH session. In the case of HTTP, n_0 is instead the first packet of the session: since no authentication process is involved, it is really the first packet carrying useful information about the usage context of the analyzed application protocol.

6 One-class algorithm: experimental results

In this section we report experimental results that show the effectiveness of Tunnel Hunter at detecting tunnel activities over HTTP and SSH. Here we briefly provide a description of the testbed environment we used.

We run all the experiments on the 100Base-TX link that connects the edge router of our campus network to the Internet. This network serves around one thousand users and it is composed of tens of 1000Base-TX layer-2 segments plus several 100Mb/s access links.

In all experiments the packet size s assumes values in the range $[40, 1500]$, due to the size of Ethernet frames. Although inter-arrival times between consecutive packets are not generally limited, we clamp them so that the minimum value is fixed to 10^{-7} seconds, the maximum to 10^{3} seconds. The former bound replaces null valued inter-arrival times: we set this value an order of magnitude below the capturing-device precision that in our case is fixed by *Tcpdump* [34] to 10^{-6} seconds. The latter bound follows from the fact that we do not consider meaningful inter-arrival times greater than 10^{3} seconds. Since we apply a log transformation we get that variable $\log_{10}(\Delta t)$ ranges inside $[-7, 3]$. Variable s is quantized with step 1 bytes, inter arrival time with step 10^{-2}: with these rules we get $N(s) = 1461$ and $N(\Delta t) = 1001$. It is evident that the evaluation of the marginal densities DEs is straightforward: each matrix, in fact, takes around 1.5 million cells, allowing us to implement the algorithm also on embedded hardware.

In the training phase, we started by collecting a large number of non-tunneled sessions for both HTTP and SSH, by running Tcpdump on the edge gateway of our campus network. The technique we used to filter out non-tunneled sessions depends on the protocol and is described below: this phase give us the two training sets T_s and T''_s. At the same time we collected a large number

of tunneled sessions and grouped the captured flows depending on the upper application, such as POP3, SMTP, etc. Following the procedure reported in Section 5.1.2 we used T_s to gather the fingerprint of the application protocol in its target usage context (non tunneling) and T''_s to calibrate the free parameters of the system, according to the procedure described in Section 5.2.2.

After an appropriate fingerprint has been installed, as a new flow comes, its anomaly score versus the fingerprint is calculated packet by packet. At any time, starting from the n_0-th pair, the score can be compared to the threshold. We will see in the following what combination of parameters is best in our scenario for detecting tunneled traffic, while guaranteeing a low rate of flows incorrectly classified as non-legitimate.

6.1 The HTTP case

We collected semantically valid HTTP traffic crossing the campus gateway for a week, gathering a training set that we used to derive the HTTP fingerprint. Since only legitimate HTTP traffic should be included in the fingerprint, we took care of avoiding spurious flows – tunneling or P2P ones – to enter the training set, using the following simple methodology. We started by saving one 300MB long trace every hour: each trace was composed of connections originated from our network and addressed to servers on the Internet, with TCP destination port 80. We then extracted all the destination addresses and sorted them by number of visits. Finally, we opened HTTP sessions to each of the most visited sites, verifying by hand which ones were serving valid HTML pages. Once we collected three hundred verified addresses, we randomly selected from the saved 300MB traces twenty thousand flows connecting to such addresses. These flows were used to gather the training sets T_s and T''_s: we then used the former to build the HTTP fingerprint, the latter to optimize the classifier's free parameters as explained in the previous section.

We then started running a HTTP tunnel, using the package available at [22] and described in Section 3.2. We behaved like someone trying to circumvent a network policy that allows only HTTP traffic to pass through the gateway. We run one end of the tunnel on a Mac OS X workstation inside the campus network. The other end was in turn located inside the network of the Engineering faculty of the University of Rome Tor Vergata, in the network of the University of Trento, on a data center in Utah (USA), and finally on a home PC connected to Internet through fast, 8Mb/s DSL line.

In order to be able to tunnel P2P traffic as well, we setup an additional tool at the tunnel entry point. In fact, since peer-to-peer protocols such as BitTorrent are designed to simultaneously open a large number of connections towards

other peers, the actual destination “server” continuously changes in this case. To make sure to convey all peer-to-peer connections through the HTTP tunnel, we implemented a gateway on a Linux workstation inside the campus network. We then configured a few machines, dedicated to P2P activities, to use it as default gateway: their outgoing traffic was then encapsulated by this gateway inside the tunnel. The tunnel exit point provided traffic de-encapsulation and masquerading through *iptables* [35] so that backward traffic was also crossing the tunnel.

Several colleagues helped us and used the HTTP tunnels as a transport-level connections for Chat, SMTP, POP3 and P2P (BitTorrent) traffic for several weeks, so that we could collect every tunneled session at the campus edge gateway. This was simple to do since we knew a priori the addresses of the tunnel exit points.

In a few weeks we collected about seventeen thousand tunneled sessions, divided among the four protocols mentioned above as described in Table 1. At the same time, we collected another fifteen thousand legitimate HTTP flows, this time not to be used for fingerprinting. The purpose of this second set of legitimate, non-tunneling HTTP flows is to evaluate a-posteriori the ability of the classifier to detect tunneled traffic while letting actual HTTP pass. Similarly to the training set, in order to make sure that this second set was composed of legitimate HTTP traffic (as opposed to tunneling sessions), we built it by using about six thousand connections to the “verified addresses” defined above. We also added one thousand flows [1] directed to other addresses to make sure that this set would not be too similar to the training set. Finally, we included an extra eight thousand non-tunneling HTTP connections to transparent proxies located at the tunnel end-points. This last subset would serve as confirmation that the accuracy of the classifier is not related to the location of the tunnel end-point: the fact that our mechanism would correctly detect as non-tunneling HTTP even the sessions run though the proxy co-located with the tunnel exit would validate this point. The numerical results shown below support this hypothesis. The *evaluation set* was composed by the union of the tunneling sessions plus this second, legitimate HTTP set.

We now present numerical results obtained by configuring the classifier with the following configuration of the free parameters, obtained by following the optimization procedure described in the previous section. We considered a Sech kernel with $h = 1$, the first ten $(s, \Delta t)$ pairs of F_{client} flow (the classifier takes its decision after seeing at max the 11-th packet of each F_{client} flow – see Section 5.2), and the threshold T_{acc} set to 7.84.

As shown in Table 1, the classifier is able to detect almost all tunneled ses-

[1] We verified by hand that these flows were actual HTTP traffic.

Protocols	Hit ratio	# sessions
HTTP	98.71%	15000
POP3 over HTTP	100%	6400
SMTP over HTTP	100%	3500
CHAT over HTTP	100%	5900
P2P over HTTP	88.09%	1000

Table 1
HTTP sessions correctly classified (optimal parameters) and cardinality of the evaluation set.

sions, no matter what protocol is being tunneled. Furthermore, and possibly even more important, the classifier achieves about 99% accuracy in detecting legitimate HTTP traffic as opposed to HTTP tunnels. This is a very important result: it means, in fact, that the classifier would incorrectly block only 1.29% valid HTTP sessions (false-negatives), while still detecting all tunnels in at least three out of four cases. At the same time, the technique is very effective in detecting tunneling sessions: even in the worst case, Tunnel Hunter detects almost 90% of the tunneled P2P traffic.

Note that the inclusion in the HTTP evaluation set of one thousand flows *not* connecting to the "verified addresses" and eight thousand flows directed to proxies installed on the tunnel exit points ensures that the high accuracy of the classifier is not due to the statistical similarity of the HTTP evaluation set to the training set or to the location of the tunnel exit points, but to the effectiveness of the fingerprint in expressing the basic properties of valid HTTP traffic.

6.2 The SSH case

As described earlier, the aim of Tunnel Hunter in case of SSH is to detect (block) SSH sessions that are being used to tunnel other application-layer protocols: SSH sessions used for remote command execution and secure copy, instead, must pass through.

We collected traces of legitimate SSH sessions by running Tcpdump on the campus edge gateway while several colleagues helped us realizing interactive sessions and securely copying files to the same remote servers used as tunnel exit points in Section 6.1. The SSH clients used in this phase were run on a mix of different operating systems such as Mac OS X, Linux and various versions of Microsoft Windows. We kept the captures related to remote administration separated from the secure copy ones.

As described in Section 3.3, all SSH sessions perform a two-step authentication stage. In this work we assume that all legitimate, non-tunneling SSH sessions use a uniform, two-round-trip user authentication phase. This makes the classifier select the third F_{client} packet after it sees the `SSH_MSG_NEWKEYS` message as the index to the n_0 pairs. Any client based on OpenSSH will generate a two-round-trip user authentication phase when using public-keys as credentials, as would many other SSH implementations, so this choice was reasonable at least in our environment. Clearly, other network administrators might want to fix n_0 to other values, depending on the type of SSH user authentication they need to support.

Having to fix an a-priori value for n_0 has an implication on the behavior of Tunnel Hunter that needs to be pointed out. In case of authentication errors, a further attempt will result in a connection that, deviating from the pre-set statistical behavior, is going to be blocked by the classifier. However, this should not be a major problem: simply, if users are entitled to connect, they will try again from scratch. We configured all the clients to respect the assumptions described above: public-key as the user-authentication method, public-key verification enabled. We also configured clients and servers to their default value with respect to data compression, i.e., no compression of encrypted data. Users not conforming to these policies would see their legitimate SSH traffic blocked by Tunnel Hunter, which once again seems like a reasonable restriction that a network administrator could set.

After several weeks, we obtained the *training sets* T_s and T''_s, i.e., around four thousand legitimate SSH sessions, from which we derived the SSH/SCP fingerprint and performed the optimization procedure outlined in Section 5.2.2.

Protocols	**Hit ratio**	**# sessions**
SSH	99.00%	600
SCP	99.10%	1700
POP3 over SSH	88.55%	2360
SMTP over SSH	99.92%	4300
CHAT over SSH	90.69%	2100
P2P over SSH	82.45%	1600

Table 2
SSH sessions correctly classified (optimal parameters) and cardinality of the evaluation set.

As we did for the HTTP case, we then collected traffic for the *evaluation set*. We recorded about another six hundred interactive sessions and about one thousand and seven hundred bulk-transfer sessions. We also recorded encrypted flows tunneling Chat, POP3, SMTP and P2P protocols. To achieve this

task, we replicated the methodology used in Section 6.1, obviously changing to SSH port-forwarding the kind of tunneling protocol. Once again, we excluded the host authentication phase from every session: this let us filter out from the evaluation set all the spurious sessions that did not complete host authentication. Table 2 reports the number of the sessions that finally composed the SSH evaluation set, as well as the best results we obtained. They were achieved configuring the algorithm to work on F_{client} flows, hyperbolic secant kernel with $h = 1$, threshold value fixed to 6.86, and computing the anomaly score in the window $[6 \rightarrow 13]$.

As the results suggest, Tunnel Hunter can detect legitimate SSH/SCP sessions with more than 99% accuracy [2], leading to a false-negative ratio lower than 1%: the vast majority of SSH and SCP sessions are not blocked by the classifier, which is a rather important result.

At the same time, Tunnel Hunter can block a very significant fraction of the tunneled flows. SMTP sessions are detected with accuracy very close to 100%. Similar performance are not reached in the other cases, although the worst classification result (i.e., P2P) is still quite good, going over 82%. In the following section we will show how the accuracy of the system can be further improved, by increasing the system's knowledge about the outlier classes.

6.3 Performance considerations

Together with accuracy, completeness is a crucial parameter to evaluate the performance of Tunnel Hunter. In the literature, completeness is usually referred to as the percentage of traffic that a classifier can consider, regardless of the correctness of its decisions. In this sense, completeness is indeed a primary index of the actual applicability of a classification technique: reaching a high accuracy makes little sense if a classifier cannot deal with the vast majority of traffic.

In the case of HTTP, using a shifted window (see Section 5.2.2) with n_0=1, the completeness of Tunnel Hunter is exactly of the 100%: as many flows come to the classifier, as many classification verdicts we have.

In the case of SSH, we fixed n_0 at the sixth packet after the host-authentication stage ends. This means that any SSH session that would be terminated before the end of the user authentication phase cannot be taken into consideration by Tunnel Hunter. Therefore, completeness in the case of SSH traffic can theoretically be lower than 100%. However, for the purposes of detecting tunneling

[2] Weighing the interactive SSH and SCP cases according to the size of evaluation sets.

flows, the fraction of SSH flows that are not considered by the classifier are not important at all, since any session that does not go beyond the user authentication phase cannot be used for transporting any other protocols.

Aside from the issue of fingerprinting, the way the classification algorithm works indicates that the underlying engine could be straightforwardly implemented in hardware based on any of the current integrated-circuit technologies. Since each protocol description is nothing more than a look-up table, the algorithm implementation requires as much memory as needed to store the desired number of fingerprints, i.e., look-up tables. The computation of the anomaly score of a flow is as simple as an adder. The classification of a flow, in fact, is reduced to the algebraic sum of n values obtained by mapping the flow to n PDF masks associated to each fingerprinted protocol: this operation is then repeated for all the fingerprinted protocols. Preliminary tests confirm that even a mid-level general purpose computer could effectively run the tunnel detector on a 100 Mb/s network interface. High-end computers with PCI Express interfaces could manage higher speeds, up to one Gb/s.

7 Multi-class classification: fingerprinting the anomalies

The algorithm described in the previous sections can be seen as a *blind approach* to the detection of legitimate traffic: the starting point, in fact, is to provide the system with as little information as possible. We characterize only the *acceptance target class* ω_t, and train a classifier to detect just that kind of traffic. Everything not conforming to ω_t belongs implicitly to the *rejected class* ω_r which is not even known to the classifier. In this sense, the described approach represents the lowest possible bound for trained classification approaches, since it is trained only with legitimate traffic.

In this section we show that Tunnel Hunter can perform better if it is provided with more knowledge about the nature of the traffic that needs to be blocked. It goes without saying that it would be practically impossible to **accurately describe** the non-legitimate class, as this would require the characterization of every possible non-legitimate usage of either HTTP or SSH: there is no practical bound on the number of applications that can be tunneled on top of these two protocols. However, one could begin by characterizing a small subset of non-legitimate (tunneling) activities, thereby adding more knowledge on the outlier classes.

This is a problem that is quite commonly referred to in the literature, where there are a well-defined target class ω_t and a poorly defined (or ill-defined) outlier class ω_o: this new class collects only a few sample subclasses describing patterns that do not belong to ω_t. According to this model, we train the

classifier on both ω_t and ω_o and we define a rejection schema that can decide when a new pattern should be, assigned to ω_r or to ω_o, instead of being assigned to the target class. Since the goal of Tunnel Hunter is ideally to block all not-legitimate traffic, we do not make any distinction between a pattern assigned to ω_r or to ω_o, and reject all flows classified as belonging to these two classes. The rationale behind this approach is that adding classes can reduce the number of cases where the uncertainty - typical of the one-class approach - could assign to the target class a pattern that should have been rejected. We will briefly discuss about this topic later.

We performed the experiments with multi-class versions of Tunnel Hunter on SSH traffic. The key idea is to characterize the profiles of a few of the possible usages of this protocol: remote administration and remote copy define the target class; tunneling a few known application protocols defines a few outlier classes. We choose SSH rather than HTTP for two main reasons. First, the accuracy of the one-class technique in the SSH case is not as good as in the HTTP case. Second, HTTP traffic is in plain-text and if need be, deep packet inspection technique could be settled to improve classification results. Instead, due to encryption, no other mechanisms can help the classification of SSH traffic, thus making its detection more challenging.

7.1 Multi-class classification

We show here how to generalize the one-class algorithm to multi-class case. Given an unknown flow F, the algorithm has to compute the anomaly score for each of the characterized classes as follows:

$$S(\vec{x}|\omega_i) = \left| \frac{\log_{10} \prod_{j=n_0}^{n} p(x_j|\omega_i)}{(n - n_0 + 1)} \right|, \tag{7.1}$$

where i indexes the corresponding target or outlier classes. The algorithm has to decide whether F is conforming to at least one of the known classes or to none of them: this last case corresponds to say that F belongs to the rejected region ω_r. A Maximum Likelihood decision function assigns the flow to the candidate class and a classification verdict is emitted based on the following algorithm:

(1) compute the anomaly scores of the flow F against all the available classes and select the candidate class ω_m so that:

$$S(\vec{x}|\omega_m) = \min_{\omega_i \in C} \{S(\vec{x}|\omega_i)\},$$

where C is the set of available classes (target and outlier);

(2) assign F to the class ω_t whenever both the following conditions are satisfied and reject otherwise:
- $\omega_m \equiv \omega_t$ (i.e., the candidate class is ω_t),
- $S(\vec{x}|\omega_m) \leq T_{acc}$,

where T_{acc} is the acceptance threshold computed in Section 6.2.

In [20], the authors described how the combination of outlier classes and a rejection algorithm based on thresholds can improve the classification results. A threshold is associated with each class (target and outlier) and the pattern belongs to the candidate class if the test with the related threshold is successful. This allows the system to assign the pattern to ω_o rather than ω_r. In our case, we do not need to make such a distinction: our goal is to block all not-legitimate sessions, regardless of their actual nature. Therefore, we can limit ourselves to computing only the threshold related to ω_t: if the candidate class is ω_o, the flow will be rejected anyway. Moreover, the threshold exactly corresponds to T_{acc}, thus making the mechanism tunable on a predefined target ratio and modular to the introduction of new ω_o.

7.2 Experimental results

We run the experiment on the SSH traffic, by training the system to recognize some tunneling protocols. We gradually increase the knowledge of the system as depicted in Figure 3 and according to the "divide et impera" principle.

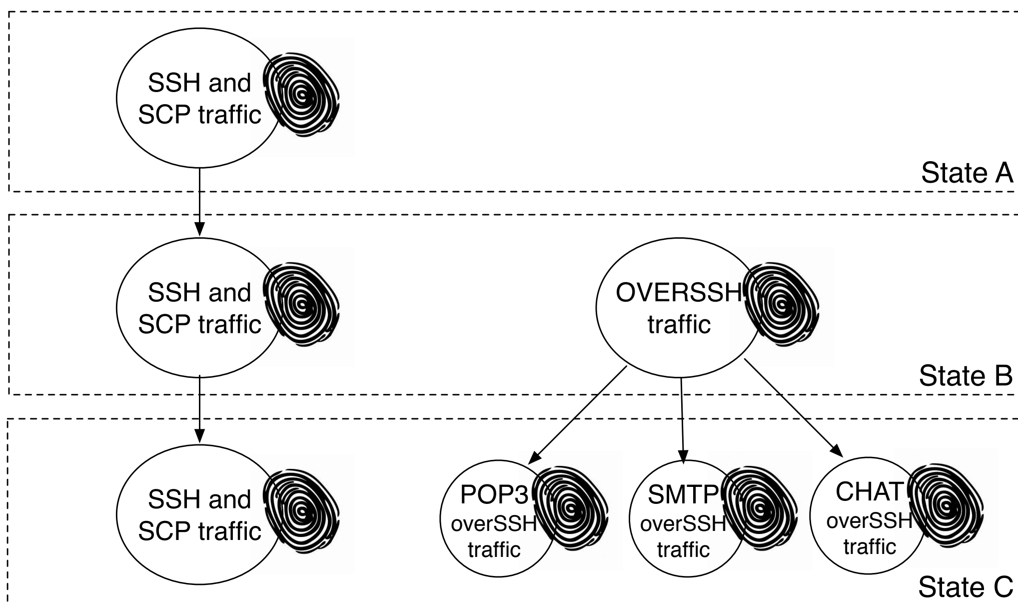

Fig. 3. Adding knowledge to the Tunnel Hunter: the "divide et impera" principle.

We have described state A in Section 6, when we trained the Tunnel Hunter for

considering only the legitimate traffic generated by SSH and SCP. No outlier class was introduced in this state.

In state B we add an outlier class, composed of a set of tunneled-sessions of POP3, SMTP and CHAT protocols. From this class we derive the fingerprint of the OVERSSH outlier traffic. We isolate this set before beginning any classification, and take care not to include any of the traces used in the evaluation phase. From here on we do not include in the fingerprints the P2P tunneled traffic: we consider such traffic as not-fingerprinted one, so that we can test the ability of the technique in rejecting unknown traffic. In other words, we use the (tunneled) P2P traces as proof that a system trained for detecting the target class, as well as a *limited number* of outlier classes, can still be effective at classifying other traffic that does not belong to any of the classes present in the training set. In our experiments, tunneled P2P traffic plays exactly this role.

In state C we split the fingerprint of OVERSSH traffic into its basic components, thus gathering one outlier fingerprint for each kind of tunneled traffic, POP3, SMTP and CHAT respectively.

In each experiment, we leave unchanged the n_0 parameter and the T_{acc} threshold value. We gather the fingerprint for each new outlier class by following the procedure described in Section 5.1.2 and evaluating the optimal kernel function as described in Section 5.2.2.

The classification results and the optimal parameters are reported in Tables 2 to 4 (Table 2 refers to the classification results related to the state A). The first important outcome resulting from these experiments is that the hit ratio of legitimate traffic remains very high as the knowledge of the system increases, i.e., going from state A to state C: this confirms that the parameters are tuned accurately. Moreover, fingerprinting the tunneled traffic results in a lower ratio of tunneled sessions labeled as legitimate. In other words, false-positives decrease as more knowledge is added to the system.

Let us stress an important point here: note how the accuracy in detecting tunneled P2P sessions improves as the classifier moves from state A to state C, going from 88.09% to 99.79%. This shows how adding knowledge on the outlier classes can positively affect the accuracy of Tunnel Hunter in detecting *all non-legitimate flows*, even those not included in the fingerprinted ω_o classes.

In Figure 4 (on the left) is depicted the trend of hit ratio as the knowledge of the system increases. We refer to the hit-ratio of legitimate traffic as the percentage of SSH and SCP traffic detected as legitimate. In each state we compute the weighing mean of hit-ratio of the SSH and SCP traffic and we report it on the chart. The hit-ratio of non-legitimate traffic is instead the percentage of tunneled session detected as not-legitimate by our system and

Protocol	SSH and SCP	Unknown
SSH	99.00%	1.00%
SCP	99.10%	0.90%
POP3 over SSH	0.66%	99.34%
SMTP over SSH	0.08%	99.92%
CHAT over SSH	0.08%	99.92%
P2P over SSH	0.47%	99.53%

Table 3
Classification results: state B. Multi-class classification: one fingerprint for SSH and SCP traffic and one for POP3, SMTP and CHAT over SSH traffic. A hyperbolic secant kernel function with $h = 1$ was used for the ω_o class.

Protocol	SSH and SCP	Unknown
SSH	99.33%	0.67%
SCP	98.91%	1.09%
POP3 over SSH	0.28%	99.72%
SMTP over SSH	0.00%	100.00%
CHAT over SSH	0.00%	100.00%
P2P over SSH	0.21%	99.79%

Table 4
Classification results: state C. Multi-class classification: one fingerprint for SSH and SCP traffic and one for each protocol over SSH, that is POP3, SMTP and CHAT traffic respectively. A hyperbolic secant kernel function with $h = 3$ was used for each ω_o class.

we compute it as in the legitimate case. As we can see, the state C leads to a precision of about the 100% in blocking non-legitimate sessions, maintaining a precision near to 99% in recognizing SSH and SCP traffic.

In Figure 4 (on the right) we also report the best trend of hit-ratio when a Gaussian kernel is used, T_{acc} is tuned accordingly and n_0 is left unchanged. As the comparison suggests, we achieve better results if a hyperbolic secant kernel is used during density estimations (about 1% on average). However, the choice of the kernel function is not unique. In this work we compared two kernel functions among others: the difference in performance proves that the kernel function is a key parameter that should be considered during the optimization phase.

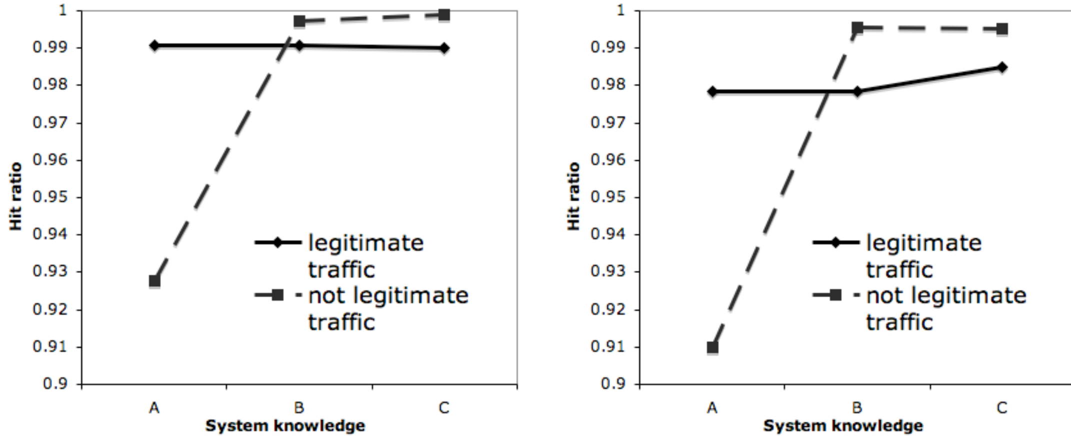

Fig. 4. Adding knowledge to the system: hit-ratio trend. On the left, optimal results by using a hyperbolic secant kernel. On the right, optimal results by using a Gaussian kernel.

8 Discussion on potential attacks

The technique recognizes with good accuracy the presence of HTTP or SSH tunneling activity after the first few packets of a TCP session are exchanged. This is certainly a benefit referred to a real-time use of the technique: few state information have to be stored before a classification verdict is emitted. Once the decision is taken, the session is labeled and processed accordingly: it will pass if carrying authorized traffic, otherwise it will be blocked.

The results shown in Section 6 suggest that Tunnel Hunter is a very precise mechanism to detect non-tunneled traffic. This proves that the class description we provide is very accurate and distinctive for the legitimate behavior of the HTTP and SSH protocols. The results presented in Section 7 show that the accuracy can be improved adding knowledge to the classification process. This means that if we can characterize different behaviors of the same application protocol, we can tune the degree of hit and false positives ratio, leading to an accuracy of nearly 100% both in blocking tunneled traffic and in detecting legitimate traffic.

As we discussed extensively in Section 6.2, the information given by the first packets could not be completely reliable to discriminate the actual application protocol carried by a TCP session. Let us consider the case of a malicious user that opens a tunnel over SSH and initially uses it for remote administration. After this allowed stage, he starts exploiting the session to tunnel other kinds of application protocol. Since the first packets are legitimate (interactive) SSH traffic, our classifier will label the session as authorized, letting it pass throughout the edge router and definitely making the user circumvent the network security policies. Currently, the technique is vulnerable to this attack

since it lacks the ability to perform recurring checks on long-term connections. We plan to further investigate this aspect, and verify the performance of the model when several "temporal windows" are considered in the decision process.

The technique suffers from another problem, i.e., it is sensitive to packet-size and timing value manipulation. As a matter of fact, this is a very serious issue for any statistical classification approach: some projects, like the NetCamo (Network Camouflaging) one [36], are just working in this direction to defeat statistical analysis. A motivate user could spend some time tracing the statistic of, for instance, actual HTTP traffic; he could then design a tunneling tool working on top of HTTP that generates packets according to such statistics, by controlling the delay between packets or by opportunely padding the packet lengths. The objective would be to conform the behavior of a tunneled session with that of a legitimate one, hence bypassing once again the network security policy.

The analysis of the portability of the fingerprints from one network to another one could help in evaluating the robustness of the technique towards this kind of attack. At this stage of development, the technique requires the HTTP and SSH fingerprints to be derived from traffic traversing the same node that performs the classification. Although the filtering with kernel functions should make the fingerprint description resistant to noise effects, we have not yet tested the portability of fingerprints from a node to another. The variables on which the fingerprints are built depend on the network conditions, i.e., link capacity and congestion levels, and we expect that the fingerprint description of the same protocol may differ if the traffic is captured on different networks. An analysis of the significance of this discrepancy could shed some light on the efforts that an attacker needs to make for bypassing the detector. The research of countermeasures for such an attack is quite challenging and in the future we plan to examine more carefully the effects that padding, fragmentation and packet delays have on Tunnel Hunter.

Nevertheless, this paper also points out a privacy vulnerability of the SSH protocol. In spite of the encryption mechanism and by looking only at the IP-level information, we have chances to infer if a user is performing remote administration or if he is involved in tunneling activities. According to this, further implementations of the SSH protocol should take into account this vulnerability and adopt countermeasures, for instance by generating only packets of the same size. At this stage of development, we presume that such countermeasure, quite easy to implement, could heavily reduce the contribution of the feature "packet size" to the fingerprint description, and therefore succeed in preserving the privacy of users, albeit with some costs in terms of bandwidth.

9 Conclusions and future work

In this paper we have presented a statistical classification mechanism called Tunnel Hunter that can successfully recognize whenever a generic application protocol is tunneled on top of HTTP or SSH. In its general form, the method provides a behavioral description of an application protocol, that we named fingerprint, by considering three simple properties of IP packets: their size, inter-arrival time and arrival order. After a training phase, the fingerprint is intended to enclose the legitimate behavior of the target application protocol, HTTP or SSH in our case, and detect whenever tunnel activities run on top of it.

The system provides parameters that can be configured to obtain any pre-set goal with respect to the desired accuracy figures: Tunnel Hunter can be configured to achieve nearly zero false-negative rates (no legitimate flows are blocked), while obtaining very high detection rates for tunneled traffic. Alternatively, system administrators could decide to further increase the detection rate for tunneling traffic to nearly 100%, by slightly increasing the false-negative rates.

Moreover, we showed how increasing the knowledge of the system with respect to tunneled traffic can significantly improve its performance, while still giving network administrator a manageable system with respect to the complexity of the training operations.

The experimental results we obtained are very promising. First and foremost, virtually no legitimate traffic is blocked by our mechanism, since over 99% of valid HTTP and SSH/SCP sessions are correctly recognized as legitimate by Tunnel Hunter. Second, and equally important, the vast majority of tunneled traffic is blocked by the mechanism. Last, hardly any sessions that come to the classifier are discarded, resulting in a completeness of the technique near to 100% (exactly 100% in the HTTP case).

These results suggest that this technique can be used to complement existing ALGs in two different ways. On one hand, it can augment their ability to recognize outgoing tunneled traffic, even if encrypted, therefore improving the effectiveness of security policies against non-complying users. On the other hand, it can help anti-virus and anti-spyware tools in detecting outgoing tunneled traffic generated by robots and compromised machines. Although this paper reports only results related to the first class of applications, we plan on continuing our tests to verify the applicability of this technique to the second class.

The next natural step is to improve the model, by introducing new variables or by studying the class-conditional densities, hopefully leading to a more ac-

curate description of the fingerprint. Moreover, we will continue to investigate the role of each variable in the model itself, for example by studying more precisely the weight of packet-size and inter-arrival time in determining the accuracy of the classification results. Finally, we are actively studying if and how higher-order features could make the fingerprints resistant to active manipulation, for example to the packet size, designed to circumvent the tunnel detector.

References

[1] J. Hill, Bypassing Firewalls: Tools and Techniques, in: 12th Annual FIRST Conference, Chicago, IL, USA, 2000.

[2] T. Ylonen, C. Lonvick, The Secure Shell (SSH) Protocol Architecture, RFC 4251, IETF (Jan. 2006).

[3] M. Crotti, M. Dusi, F. Gringoli, L. Salgarelli, Traffic Classification through Simple Statistical Fingerprinting, ACM SIGCOMM Computer Communication Review 37 (1) (2007) 5–16.

[4] M. Crotti, M. Dusi, F. Gringoli, L. Salgarelli, Detecting HTTP Tunnels with Statistical Mechanisms, in: Proceedings of the 42th IEEE International Conference on Communications (ICC 2007), Glasgow, Scotland, 2007, pp. 6162–6168.

[5] M. Dusi, M. Crotti, F. Gringoli, L. Salgarelli, Detection of Encrypted Tunnels across Network Boundaries, in: Proceedings of the 43rd IEEE International Conference on Communications (ICC 2008), Beijing, China, 2008.

[6] K. Borders, A. Prakash, Web tap: detecting covert web traffic, in: CCS'04: Proceedings of the 11th ACM conference on Computer and Communications Security, Washington DC, USA, 2004, pp. 110–120.

[7] G. Bissias, M. Liberatore, D. Jensen, B. N. Levine, Privacy Vulnerabilities in Encrypted HTTP Streams, in: Proc. Privacy Enhancing Technologies Workshop (PET 2005), Dubrovnik, Croatia, 2005.

[8] M. Liberatore, B. N. Levine, Inferring the source of encrypted http connections, in: CCS '06: Proceedings of the 13th ACM conference on Computer and Communications Security, Alexandria, Virginia, USA, 2006, pp. 255–263.

[9] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, P. Tofanelli, Revealing skype traffic: When randomness plays with you, SIGCOMM Computer Communication Review 37 (4) (2007) 37–48.

[10] C. V. Wright, F. Monrose, G. M. Masson, On Inferring Application Protocol Behaviors in Encrypted Network Traffic, Journal of Machine Learning Research 7 (2006) 2745–2769.

[11] V. Paxson, Bro: a system for detecting network intruders in real-time, Computer Networks 31 (23–24) (1999) 2435–2463.

[12] M. Roesch, SNORT: Lightweight Intrusion Detection for Networks, in: LISA '99: Proceedings of the 13th USENIX Conference on Systems Administration, Seattle, WA, USA, 1999, pp. 229–238.

[13] V. Paxson, Empirically derived analytic models of wide-area TCP connections, IEEE/ACM Transactions on Networking 2 (4) (1994) 316–336.

[14] F. Hernández-Campos, F. D. Smith, K. Jeffay, A. B. Nobel, Statistical Clustering of Internet Communications Patterns, in: Computing Science and Statistics, Vol. 35, 2003.

[15] A. McGregor, M. Hall, P. Lorier, J. Brunskill, Flow Clustering Using Machine Learning Techniques, in: Proceedings of the 5th Passive and Active Measurement Workshop (PAM 2004), Antibes Juan-les-Pins, France, 2004, pp. 205–214.

[16] M. Roughan, S. Sen, O. Spatscheck, N. Duffield, Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification, in: IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, Taormina, Sicily, Italy, 2004, pp. 135–148.

[17] A. W. Moore, D. Zuev, Internet traffic classification using bayesian analysis techniques, in: SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Banff, Alberta, Canada, 2005, pp. 50–60.

[18] C. Chow, On optimum error and reject tradeoff, IEEE Transactions on Information Theory 16 (1) 41–46.

[19] T. C. W. Landgrebe, D. M. J. Tax, P. Paclík, R. P. W. Duin, The interaction between classification and reject performance for distance-based reject-option classifiers, Pattern Recogn. Lett. 27 (8) (2006) 908–917.

[20] G. Fumera, F. Roli, G. Giacinto, Multiple reject thresholds for improving classification reliability, Lecture Notes in Computer Science 1876.

[21] F. Heinz, J. Oster, DNS tunnel, http://nstx.dereference.de/nstx.

[22] R. Mills, The Linux Academy HTTP Tunnel, http://the-linux-academy.co.uk/downloads.htm.

[23] L. Brinkhoff, GNU httptunnel, http://www.nocrew.org/software/httptunnel.html.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, IETF (Jun. 1999).

[25] OpenSSH, http://www.openssh.org.

[26] T. Ylonen, C. Lonvick, The Secure Shell (SSH) Authentication Protocol, RFC 4252, IETF (Jan. 2006).

[27] T. Ylonen, C. Lonvick, The Secure Shell (SSH) Transport Layer Protocol, RFC 4253, IETF (Jan. 2006).

[28] A. Webb, Statistical Pattern Recognition, 2nd Edition, Wiley, 2002, iSBN 0-470-84514-7.

[29] R. O. Duda, P. E. Hart, Pattern classification and scene analysis, Wiley-Interscience, New York, 1973.

[30] V. N. Vapnik, Statistical learning theory. Adaptive and learning systems for signal processing, communications, and control, Wiley, New York, 1998.

[31] L. Devroye, L. Gyorfi, G. Lugosi, A Probabilistic Theory of Pattern Recognition, Springer, New York, 1996.

[32] P. Domingos, M. Pazzani, On the optimality of the simple bayesian classifier under zero-one loss, Machine Learning 29 (2-3) (1997) 103–130.

[33] D. Tax, R. Duin, Uniform object generation for optimizing one-class classifiers (2002).

[34] Tcpdump/Libpcap, http://www.tcpdump.org.

[35] N. C. Team, netfilter/iptables, http://www.netfilter.org.

[36] T. A. U. Department of Computer Science, Network Camouflaging, http://research.cs.tamu.edu/netcamo.