HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows

Xiapu Luo[§*], Peng Zhou[§], Edmond W. W. Chan[§], Wenke Lee[†], Rocky K. C. Chang[§], Roberto Perdisci[‡]
The Hong Kong Polytechnic University[§], Georgia Institute of Technology[†], University of Georgia[‡]
{csxluo,cspzhouroc,cswwchan,csrchang}@comp.polyu.edu.hk, wenke@cc.gatech.edu, perdisci@cs.uga.edu

Abstract

Leakage of private information from web applications—even when the traffic is encrypted—is a major security threat to many applications that use HTTP for data delivery. This paper considers the problem of inferring from encrypted HTTP traffic the web sites or web pages visited by a user. Existing browser-side approaches to this problem cannot defend against more advanced attacks, and server-side approaches usually require modifications to web entities, such as browsers, servers, or web objects. In this paper, we propose a novel browser-side system, namely HTTPOS, to prevent information leaks and offer much better scalability and flexibility. HTTPOS provides a comprehensive and configurable suite of traffic transformation techniques for a browser to defeat traffic analysis without requiring any server-side modifications. Extensive evaluation of HTTPOS on live web traffic shows that it can successfully prevent the state-of-the-art attacks from inferring private information from encrypted HTTP flows.

1 Introduction

Leakage of private information from web applications is a major security threat to many applications that use HTTP for data delivery. Cloud computing and other similar service-oriented platforms will only exacerbate this problem, because these services are usually delivered through web browsers. Moreover, it is well known that data encryption alone is insufficient for preventing information leaks. For instance, traffic-analysis attacks can identify the web sites visited by a user from encrypted traffic [7, 22, 23, 26] and anonymized NetFlows records [14]. Chen et al. have further showed that sensitive personal information, such as medical records and financial data, could also be inferred through traffic analysis [13]. Besides, a user's browser could be fingerprinted [39], and her browsing patterns could be profiled from traffic features [29]. A common approach to preventing leaks is to obfuscate the encrypted traffic by changing the statistical features of the traffic, such as the packet size and packet timing information [13, 23, 35, 38].

Existing methods for defending against information leaks, however, suffer from quite a few problems. A major problem is that, as server-side solutions, they require modifications of web entities, such as browsers, servers, and even web objects [13, 38]. Modifying the web entities is not feasible in many circumstances and cannot easily satisfy different applications' requirements on information leak prevention. A second fundamental problem with these methods is that they are still vulnerable to some advanced traffic-analysis attacks. For example, although Sun et al. [35] proposed twelve approaches to defeat their traffic-analysis attack based on web object size, new attacks based on the tuple of packet size and direction [26] could still identify the web sites visited by a user. Finally, the efficacy of these methods has not been validated thoroughly based on actual implementations and live HTTP traffic. An exception is the work from Chen et al. [13] that is implemented as an IIS extension and a Firefox add-on.

In this paper we explore a browser-side approach to prevent information leaks from encrypted web traffic. Compared with the server-side approach, the browser-side approach has the scalability advantage, because only the traffic between the browser and the visited servers needs to be obfuscated. Moreover, it is possible for users to choose which encrypted flows to be obfuscated in order to conserve resources and to reduce impacts on performance, but this flexibility advantage is very difficult to obtain from a server-side approach. However, designing a browser-side method is very challenging, because the server's behavior cannot be directly modified to evade traffic analysis. That is, we cannot apply the previously proposed methods that assume the capability of modifying the server's behavior to the browser-side approach.

We show in this paper that it is possible to devise a browser-side method to defeat traffic analysis by presenting HTTPOS (which stands for HTTP or HTTPS with Obfus-

*Most of the work by this author was performed while at Georgia Tech.

cation), our proposed browser-side method. In addition to the advantages discussed above, HTTPOS has several other important advantages, such as supporting a wider scope of application scenarios than the previous approaches (see the threat model in Section 2). HTTPOS is a user-level program and does not need to modify any web entity. It obfuscates encrypted web traffic by modifying four fundamental network flow features on the TCP and the HTTP layers, namely, packet size, timing of packets, web object size, and flow size. These features, as shown in the evaluation, are sufficient for diffusing and confusing the existing traffic-analysis attacks. To modify the traffic from web servers, HTTPOS exploits a number of basic features in TCP (e.g., Maximal Segment Size (MSS) negotiation and advertising window) and HTTP (e.g., HTTP Range and HTTP Pipelining).

We have implemented HTTPOS and conducted extensive experiments on live HTTP traffic to evaluate its performance in terms of evading traffic-analysis attacks and impacts on the performance of network flows. The results show that HTTPOS can effectively prevent the state-of-the-art attacks from inferring private information from encrypted HTTP flows. As will be shown in Section 5.2, without HTTPOS an attacker can achieve high accuracy of inferring the web sites visited by a user. For example, some attacks can achieve 94% accuracy on inferring the 100 web sites we tested by selecting the one with the highest confidence based on their classification algorithm. With HTTPOS *all* attacks' accuracy drops to zero for at least 98 web sites. Even if an attacker chooses the top five web sites as her inference, the accuracy for at least 94 web sites remains zero for *all* attacks. Moreover, some traffic-analysis attack can easily infer a user's input to the Google search box. But when HTTPOS is applied, the output of such attack is reduced to a random guess.

Section 2 first presents the threat model, and Section 3 describes our strategies for evading traffic-analysis attacks and methods for manipulating network flow features. After that, we introduce HTTPOS's design and implementation in Section 4, followed by extensive experiment results in Section 5. We finally introduce the related work in Section 6 before concluding this paper in Section 7.

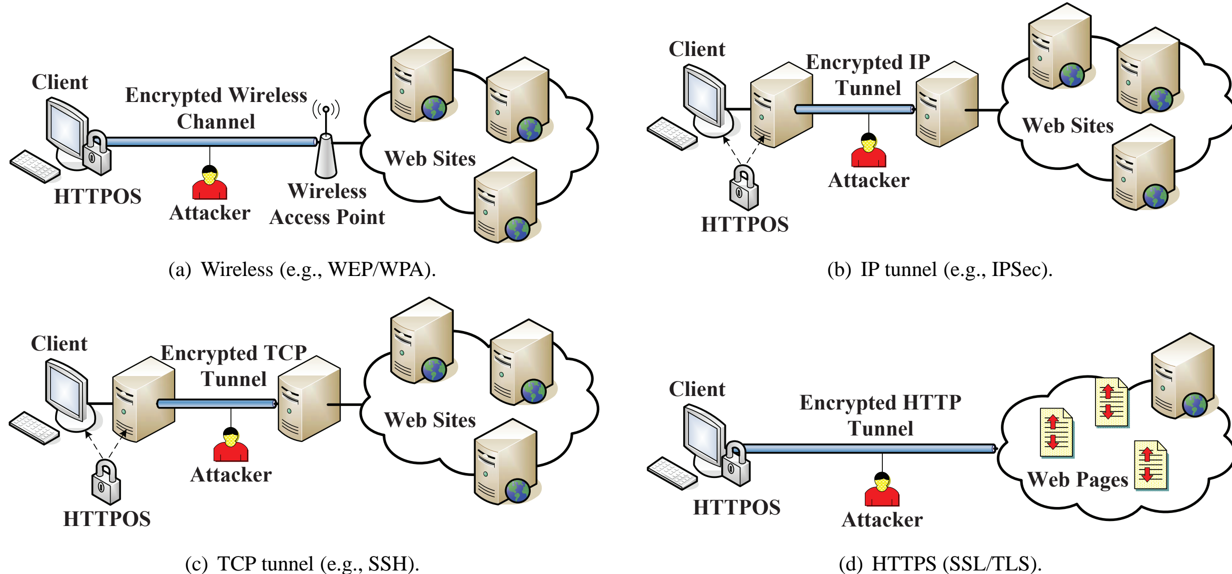
2 Threat models

Unlike previous works, we consider in this paper *both* (1) the problem of inferring the web sites visited by users and (2) the problem of inferring the web pages browsed by users. The three attack scenarios illustrated in Figures 1(a)-1(c) concern problem (1), whereas the one in Figure 1(d) concerns problem (2). We summarize the threat models for the four scenarios in Table 1 based on the attack goals, visibility of the packet information to the attacker, and HTTPOS's location. There are five entities in the threat models: a victim user (i.e., client in Figure 1), an attacker, an encrypted tunnel, HTTPOS, and remote web sites/pages. The threat models for scenarios (a)-(c) were adopted in previous works, including Sun et al. [35], Liberatore et al. [26] and Wright et al. [38], and the threat model for scenario (d) was adopted by Chen et al. [13].

In scenarios (a)-(c), a client visits a web site through an encrypted tunnel at different layers, for example, wireless channel with WEP/WPA [21], IPSec-based IP tunnel [37], and SSH-based TCP tunnel [6], and the attacker attempts to find out that web site. In scenario (d), a client visits different web pages at a certain web site. This attack model assumes that the attacker knows the web site, and she attempts to discover the web pages visited by the client. Note that an updated web page due to the client's interactions with the web site is considered as a new web page from an attacker's viewpoint. For example, some web sites (e.g., Google) may return auto-suggested words upon receiving a user input. These dynamically updated web pages are regarded as different web pages.

In all four scenarios, the attacker eavesdrops the encrypted tunnel to obtain the encrypted packets sent between the victim and web servers, but she cannot decrypt these packets. To infer the visited web sites/pages from these encrypted packets, the attacker first profiles the characteristics of the traffic between the victim and each candidate web site/page. The traffic profiling depends on the traffic analysis methods [7, 13, 22, 23, 26]. She can easily build such profiles by visiting those web sites/pages via the encrypted tunnels herself. Equipped with the set of traffic profiles, the attacker then performs the inference by classifying the captured traffic trace into the traffic profiles prepared beforehand. From the viewpoint of pattern classification, the traffic profiling step is known as conducting supervised learning to train a classifier, whose feature set is the traffic profile, and the class label is the web site/page. Moreover, the inference step corresponds to labeling a traffic trace according to the trained classifier [18].

HTTPOS obfuscates the encrypted traffic by exploiting the protocol features in TCP and HTTP. Since the TCP connection (and therefore the HTTP connection) is end-to-end in scenarios (a), (b), and (d), HTTPOS can be deployed at the browsers. On the other hand, the browser's TCP connection is terminated at the tunnel entry in scenario (c). Therefore, when HTTPOS is deployed at the browsers for TCP tunnels, only the HTTP methods can be used for traffic obfuscation. Furthermore, HTTPOS can be deployed at the tunnel entry in scenarios (b) and (c). Since we implement HTTPOS as an HTTP proxy (which will be discussed in Section 4), the same HTTPOS can be placed at the browser or the tunnel entry for both scenarios. However, placing HTTPOS at the TCP tunnel's entry maximizes HTTPOS's



(a) Wireless (e.g., WEP/WPA).

(b) IP tunnel (e.g., IPSec).

(c) TCP tunnel (e.g., SSH).

(d) HTTPS (SSL/TLS).

Figure 1: The four attack scenarios considered in this paper.

Table 1: Threat models for the four attack scenarios.

	Wireless (e.g., WEP/WPA)	IP tunnel (e.g., IPSec)	TCP tunnel (e.g., SSH)	HTTPS (SSL/TLS)
Attacker's goal	Identify web site	Identify web site	Identify web site	Identify web page
Visibility of HTTP header	No	No	No	No
Visibility of TCP header	No	No	Yes	Yes
Visibility of Destination IP	No	No	No	Yes
HTTPOS's location	Client	Client/tunnel entry	Client/tunnel entry	Client

obfuscation power, because, as mentioned earlier, HTTPOS at the browser cannot use TCP's protocol features to obfuscate encrypted traffic.

3 Defending against traffic-analysis attacks

In Section 3.1, we first elaborate on the classification algorithms used in the state-of-the-art traffic-analysis attacks. Then we propose strategies with formal analysis to deceive those classification algorithms in Section 3.2. In particular, we identify four basic features that can affect the information used by those traffic-analysis attacks: packet size, timing of packets, web object size, and flow size. We then propose methods in Section 3.3 to manipulate these features to evade these traffic-analysis attacks.

3.1 The state-of-the-art attacks

We consider the five traffic-analysis attacks on encrypted HTTP flows [7, 13, 26, 35] in Table 2. We name them by concatenating the first letters of the authors' names. Sun et al. proposed the SSWRPQ attack, which is the first attack that can identify web sites through the number and size of web objects [35]. Later on, Bissias et al. proposed using the inter-arrival time between packets and packet size to profile a web site in their BLJL attack [7]. Liberatore et al. exploited the tuple (flow direction, packet size) for traffic analysis and proposed two classification algorithms: *Jaccard coefficient* (JC) and *naive Bayesian classifier* (NBC) that are referred to as LL-JC attack and LL-NBC attack, respectively [26]. Most recently, Chen et al. employed a sequence of (flow direction, packet size) to infer the web pages visited by a victim in their CWWZ attack [13].

The SSWRPQ attack employs the number and size of web objects as features. Since all HTTP traffic is encrypted, an attacker cannot obtain the exact values of such features. As suggested in [24, 35], the amount of bytes from the server between two consecutive requests from the client is used to approximate the sizes of web objects. After obtaining the number of web objects and their sizes, the attack uses the Jaccard coefficient to quantify the similarity between a new trace and an existing profile.

The BLJL attack employs both inter-arrival time (IAT) be-

Table 2: **Traffic-analysis attacks studied in this paper.**

Attacks	Features	Classification algorithms
SSWRPQ [35]	The number and size of web objects	Jaccard coefficient
BLJL [7]	Inter-arrival time between packets and packet size	Cross correlation
LL-JC [26]	Tuples of (flow direction, packet size)	Jaccard coefficient
LL-NBC [26]	Tuples of (flow direction, packet size)	Naive Bayesian classifier
CWWZ [13]	Sequence of tuple (flow direction, packet size)	Sequence comparison

tween packets and packet size to profile a web site and uses cross-correlation to measure the similarity between a new trace and an existing profile. To compare a new trace's features to an existing profile, the BLJL attack computes the cross correlation of its IAT sequences and packet size sequences by:

$$R = \frac{\sum_{i=1}^{n}[(\tau_i - \bar{\tau})(s_i - \bar{s})]}{\sqrt{\sum_{i=1}^{n}(\tau_i - \bar{\tau})}\sqrt{\sum_{i=1}^{n}(s_i - \bar{s})}}, \quad (1)$$

where τ and s denote the new trace's IAT values and packet sizes, respectively, whereas $\bar{\tau}$ and $\bar{s}$ are the respective mean values of an existing profile's IAT and packet size.

The LL-JC attack employs tuples of (flow direction, packet size) as features. Let $D = \{d_1, d_2, \ldots\}$ be a set of tuples in a trace. The JC is defined as

$$S(D_{new}, D_i) = \frac{|D_{new} \bigcap D_i|}{|D_{new} \bigcup D_i|}, \quad (2)$$

where D_{new} and D_i denote the set of tuples in a new trace and that in the profile of the ith web site, respectively. The normalized S (i.e., $\overline{S}$) is used to determine to which class a given trace belongs:

$$\overline{S}(D_{new}, D_i) = \frac{S(D_{new}, D_i)}{\sum_{j \in U} S(D_{new}, D_j)}, \quad (3)$$

where U is the set of all existing profiles.

The LL-NBC attack only considers the existence of certain tuples without examining the number of tuples in a flow. This attack employs the Kernel Density Estimation (KDE) to estimate the probability density function of each tuple (i.e., considering the value of each feature) and then employs a naive Bayesian classifier to reach a decision. We use V to denote the features used in an LL-NBC attack.

The relationship between V and D is that for a given feature in V, its value is equal to zero if the corresponding tuple is not in D and KDE is not used. After using KDE, although such features may have values larger than zero, their values may be very small, depending on the parameters used in the kernel function and the location of tuples that are in D. NBC classifies V_{new} into a class V_i if and only if

$$P(V_{new}|V_i)P(V_i) > P(V_{new}|V_j)P(V_j), \; \forall j \neq i. \quad (4)$$

Based on the assumption that all attributes are independent,

$$P(V_{new}|V_i) = \prod_{k=1}^{n} P(d_k|V_i), \quad (5)$$

where d_k denotes a feature in V_i.

The CWWZ attack, unlike the LL-JC and LL-NBC attacks that only inspect individual packets, considers a sequence of packets [13] to infer user inputs to a web page. A sequence of directional packet sizes is referred to as a *flow vector*, denoted by $C = \{c_t, c_{t+1}, \ldots, c_{t+n-1}\}$, where c_{t+n-1} represents the directional packet size observed at time $t+n-1$, and n is the sequence length. Let k be the number of all available characters and k_t be the number of possible character sequences at time t, and these possible characters constitute an *ambiguity set*. After observing c_t, the attacker may know that only k_t/α_t possible inputs from the ambiguity set can produce c_t, where $\alpha_t \in [1, k_t)$ is defined as a *reduction factor*. In the next observation, the ambiguity set's size k_{t+1} is reduced from $k \cdot k_t$ to $k \cdot (k_t/\alpha_t)$. After receiving $\{c_t, c_{t+1}, \ldots, c_{t+n-1}\}$, the size of ambiguity set is reduced from k^n to $\frac{k^n}{\Pi_{i=1}^{n}(\alpha_{t+i-1})}$, where $\Pi_{i=1}^{n}(\alpha_{t+i-1})$ is referred to as *reduction power*. A victim's input can be easily recovered if an attack has large reduction power.

3.2 Two defense strategies

We propose two general strategies to deceive an attack's classification algorithm. The first one is inducing the classification algorithm to make a random guess by introducing features that have not been involved in training the algorithm. The second strategy is to confuse the classification algorithm to misclassify a trace.

3.2.1 The diffusion strategy

The SSWRPQ, LL-JC, and LL-NBC attacks implicitly assume that packet sizes (or web object sizes) observed in the training data set will appear in the testing data set (i.e., a new trace to be classified). If all packet sizes or web object sizes in a new trace never appear in the training data set, these algorithms will be forced to make a random guess.

Lemma 1 details how to evade algorithms based on the JC and NBC.

Lemma 1. *If a flow comprises a set of tuples, denoted as D_{new}, which never exist in any training set, then the LL-JC and LL-NBC attacks cannot classify this flow correctly. Similarly, if a web object size of a flow never appears in any training set, then the SSWRPQ attack cannot identify the class of this flow.*

Proof. In the LL-JC attack, $\overline{S}(D_{new}, D_i) = 0$, because D_{new} does not appear in any training set. Similarly, if the sizes of web objects does not exist in any training set, the SSWRPQ's JC becomes zero. In the LL-NBC attack, if D_{new} is totally new to the classification algorithm, $P(V_{new}|V_i) = 0$ ($\forall i,\ i \in U$). Although using KDE may allow $P(V_{new}|V_i) > 0$ due to the kernel function, we can select D_{new} whose tuples are not close to any tuples in the training set, so that $P(V_{new}|V_i) \to 0$. □

Defense mechanisms based on Lemma 1 are feasible in practice, because the packet size is dominated by a relatively small number of values [34]. In other words, we can easily find packet sizes that never appear in any training set. Figure 2 plots the CDF of the number of unique packet sizes in a flow from two data sets. The UMass data set contains packet header traces collected four times a day from February 2006 to April 2006, and the size of compressed `pcap` files is around 2.6GB [26]. This data set is used for testing the traffic-analysis attacks in [26]. The WIDE traces, on the other hand, contain *all* traffic going through the samplepoint-F of the WIDE backbone networks from 30 March 2009 to 2 April 2009, and the size of the `pcap` traces is around 433GB [10]. Since the WIDE data set contains various kinds of traffic, we extract HTTP flows that have at least five packets. We regard the packets sent to a web server as request packets, and those sent from a web server as response packets. Both figures show that in the majority of flows the number of unique packet sizes is less than 100. Moreover, the request packets usually have fewer number of unique packet sizes than the response packets.

For the CWWZ attack, if a flow vector does not occur in any training set, the attacker cannot exclude any possible input and has to consider all the ambiguity set in the next flow vector. Therefore, the reduction power is fixed to one, and the final decision is the same as a random guess. For the BLJL attack, since it uses a cross-correlation based algorithm, it makes an implicit assumption that the number of packets (i.e., n) should be the same in the training data set and the testing data set. If this assumption does not hold, R could not be computed. Thus, it is possible to defeat this attack by changing the number of packets.

3.2.2 The confusion strategy

To confuse an attack's classification algorithm, we could manipulate a flow to make its features similar to another flow's features. For instance, Lemma 2 first presents an approach to confuse the LL-JC, LL-NBC, and SSWRPQ attacks.

Lemma 2. *Let D_i and D_j be the respective sets of tuples in the profiles of sites i and j, and V_i and V_j be the respective feature sets of sites i and j used by the LL-NBC attack. If the tuples for a flow of site i become ($D_j - D_i$) (i.e., the tuples that are in D_j but not in D_i) after changing the packet sizes in the flow, the LL-JC and the LL-NBC attacks will regard the transformed flow as a flow of site j. Similarly, after changing the sizes of web objects in a flow of site i to the one in the profile of site j, the SSWRPQ attack will regard the transformed flow as a flow of site j.*

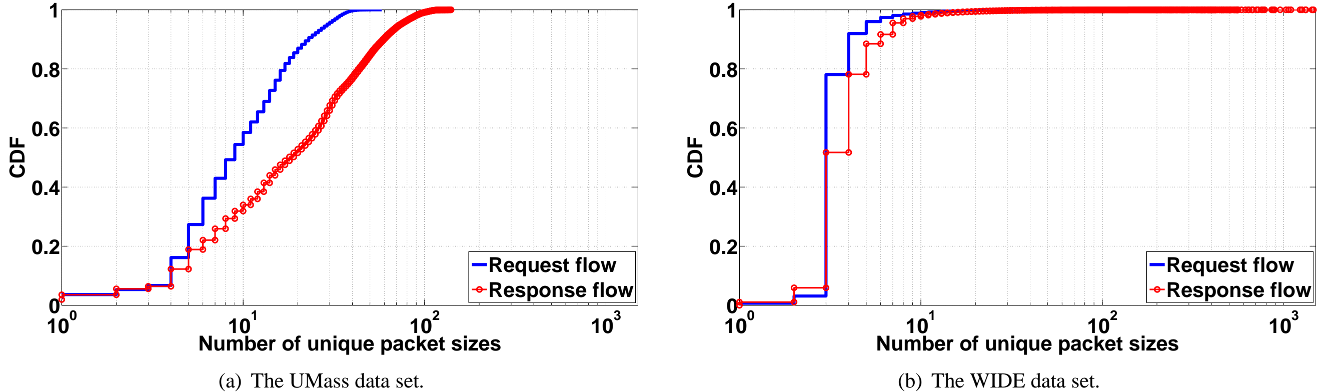
Proof. Let $\overline{D}_{new}$ be the tuples in the transformed flow. Since $\overline{S}(\overline{D}_{new}, D_j) > \overline{S}(\overline{D}_{new}, D_i) = 0$, the LL-JC attack regards the transformed flow as a flow of site j rather than a flow of site i. Let $\overline{V}_{new}$ be the feature set in the transformed flow. According to the relationship between D and V, we know that $P(\overline{V}_{new}|V_j) > P(\overline{V}_{new}|V_i) = 0$ (or $P(\overline{V}_{new}|V_i) \to 0$ if the KDE is used). Thus the LL-NBC attack will consider the transformed flow as a flow of site j. Similarly, if the transformed flow's web object size only exists in site j's profile, the SSWRPQ attack will find that the transformed flow is more similar to flows of site j than flows of site i. □

For the CWWZ attack, if we introduce other flow vectors (e.g., by entering some useless inputs) in addition to the flow vectors induced by the real inputs, the attacker has to consider all inputs that may result in these flow vectors for the following two reasons. First, the attacker could not differentiate between flow vectors caused by useless inputs and those induced by real inputs. Second, the attacker could not know the start and the end of real inputs. As a result, the reduction power can be reduced.

Lemma 3 presents a sufficient condition to induce the BLJL attack to reach an incorrect decision.

Lemma 3. *By letting all packets in a flow have the same size s_c and their IATs have the same value τ_c, R in Eq. (1) is then determined by s_c and τ_c, instead of the transformed flow's original feature.*

Proof. If $\tau_i = \tau_c$ and $s_i = s_c$, then $R = \sqrt{(\tau_c - \bar{\tau})(s_c - \bar{s})}$. By adjusting s_c and τ_c, we can therefore make a flow similar to any other flows. As a result, the BLJL attack cannot identify the original flow. □



(a) The UMass data set.

(b) The WIDE data set.

Figure 2: The distributions of unique packet sizes in two HTTP data sets.

3.3 Manipulating the features

To defeat the traffic-analysis attacks listed in Table 2 and possibly new traffic-analysis attacks, we manipulate four fundamental network flow features, including packet size, web object size, flow size, and timing of packets. Under our threat models, these features can be measured from an encrypted HTTP flow and exploited to differentiate network flows by an attacker. We describe our basic approaches for manipulating these features in HTTPOS below. Since the flow size is determined by the web object size, we discuss the flow size together with the web object size.

3.3.1 Packet size

HTTPOS alters the size of outgoing packets on the HTTP and TCP layers. On the HTTP layer, HTTPOS increases the packet size by adding additional bytes to the HTTP header, for example, adding additional fields, appending characters to the `Referer` field, or using specific media types to replace the asterisk in the `Accept` field. On the TCP layer, HTTPOS decreases the packet size by splitting a TCP packet into several smaller TCP packets. If an attacker could not observe the TCP header (i.e., scenarios (a) and (b)), HTTPOS increases the packet size by extending the TCP option fields (e.g., by adding the TCP No Operation option that is usually used to pad the option list). However, it is more challenging to control the size of incoming packets from a web server, because it is usually determined by the server application and the server's TCP/IP stack. HTTPOS exploits the HTTP Range option, TCP maximum segment size (MSS) negotiation, and TCP advertising window to manipulate the packet size.

HTTP Range HTTP/1.1 provides a Range option to fetch portion of a web object to avoid downloading the same content already received by a client [20]. RFC 2616 refers to a request having a `Range` header as a *partial GET*. To fetch a web object, HTTPOS sends a request with a `Range` header, such as "`Range: bytes=0-0`." If the server supports HTTP Range, it will reply with "`206 Partial Content`" and a `Content-range` header, such as "`Content-range: bytes 0-0/L`," where L is the actual length of the requested web object. After that, HTTPOS sends N_U requests to the web server, each of which only asks for u_i ($i = 1, \ldots, N_U$) bytes, where $\sum_{i=1}^{N_U} u_i = L$.

TCP MSS negotiation A TCP packet's payload length is limited by the TCP MSS. TCP allows sender and receiver to negotiate the MSS through the MSS option in the TCP SYN and TCP SYN/ACK packets. By exploiting this feature, HTTPOS can constrain the packet size by announcing a small MSS.

TCP advertising window Since the advertising window controls the number of bytes a sending TCP could dispatch, the size of the incoming packet can be manipulated by adjusting the advertising window if it is not larger than the MSS.

3.3.2 Web object size and flow size

It is nontrivial for a browser-side approach to affect the web object size. A possible method is to adjust the content codings [20]. For example, if a web object is compressed before being sent to the client, HTTPOS modifies the `Accept-Encoding` header to force the server to send an uncompressed web object. A similar method is to adjust the quality value [20]. Although these two methods are quite general, we find that most servers do not support them.

To disguise the size of web objects, we therefore adopt the following methods based on the observation that an attacker could only estimate the size of a web object from encrypted traffic using the amount of bytes received between two requests [24, 35].

TCP retransmissions If the client is using IP tunnel or encrypted wireless channel that prevents an attacker from seeing the TCP header, HTTPOS affects the web object size

and enlarges the flow size by causing TCP retransmissions. More precisely, since an ACK number informs a sending TCP how many bytes have been successfully received by a TCP receiver, by acknowledging only part of the received bytes, HTTPOS could force the server to re-send the unacknowledged portion along with the new portion if the packet size permits [28]. In this case, the attacker will overestimate both web object size and flow size. If the attacker can observe the TCP header, she may ignore the retransmitted packets. In this case, HTTPOS employs the following HTTP-based approaches, because an attacker cannot observe the content of encrypted packets.

HTTP Pipelining Using HTTP Pipelining to send several requests together without waiting for the corresponding responses makes it difficult for an attacker to infer the size of a web object [20], because an attacker may regard the total bytes from the server as the size of a web object. To achieve this, HTTPOS either merges several requests from the client or attaches a useless request with the original request before sending the packet to the server. In both cases, the attacker will overestimate the HTTP request's size, web object size, and flow size. When using the latter approach, HTTPOS does not forward the response to the useless request to the client.

HTTP Range Employing HTTP Range to request data that have been downloaded can enlarge the flow size and disguise the web object size. Its basic idea is similar to the method using retransmitted TCP packet. The difference is that the latter method operates on TCP and therefore requires TCP header being invisible to the attacker, whereas the HTTP header in the encrypted traffic is already invisible to the attacker. Although Sun et al. [35] also suggested using HTTP Pipelining and HTTP Range, they did not implement and evaluate them. We implement these approaches and carefully evaluate them using live HTTP traffic. We also measure the popularity of supporting HTTP Range and HTTP Pipelining (Section 4.4). Moreover, we combine the method based on HTTP Pipelining and that based on TCP advertising window to evade an advanced CWWZ attack (see Section 5.2.2).

Injecting useless requests Injecting a useless request between two requests sent by a client can disguise the web object size. More precisely, after a client sends one request, HTTPOS decreases the advertising window to a small value (say 10 bytes), so that the server cannot return all the requested content in one packet. Once a response packet is received, HTTPOS injects a random request. In this case, the attacker will underestimate the web object size, because she may observe many small responses to different HTTP requests. There is no restriction on the requests injected by HTTPOS. Its usage is to mislead an attacker into obtaining wrong web object size and flow size.

3.3.3 Timing of packets

Since the outgoing packets sent from the client go through HTTPOS, it is easy to control their timing by delaying the transmissions. To manage the timing of packets from the server, HTTPOS operates on two levels. The first level is to manipulate the timing of request packets, because the response packets are triggered by request packets. The second level is to manipulate the timing of response packets through delaying ACK packets. The rational behind this method is that without receiving ACK packets the sending TCP may not send out new data packets due to TCP's ACK-based self-clocking feature. Note that the flow sequence can also be changed by reordering the sequence of TCP SYN packets or delaying the corresponding HTTP request packets. By doing so, HTTPOS can help a user evade the traffic-analysis attack in [14].

4 HTTPOS

4.1 Design

HTTPOS acts as a proxy through which a client visits a web site. It accepts HTTP requests from the client and modifies them as needed before sending them out. Figure 3 illustrates the HTTPOS operations. When HTTPOS receives a URL, it checks whether information related to this URL is in the cache, which includes whether the URL can be fetched through HTTP Range, whether the server supports HTTP Pipelining, and the web object size. This information determines which module will be used. Each module realizes a method described in Section 4.2.

If it is the first time for HTTPOS to handle a URL, HTTPOS uses the method based on TCP advertising window (Section 4.2.1). It manipulates the advertising window in outgoing packets to control the size of response packets. To test whether the URL can be fetched through HTTP Range, HTTPOS inserts "`Range: bytes=0-0`" to the outgoing HTTP request. To verify whether the server supports HTTP Pipelining, HTTPOS duplicates the HTTP request and adds "`Range: bytes=1-1`" to it and then sends these two HTTP requests to the server at the same time. After receiving the responses, the feature information and the web object size are saved in the cache.

If the information is found in the cache, HTTPOS selects the proper method based on its effectiveness on evading traffic-analysis attacks and mitigating performance degradation. If the URL could not be fetched through HTTP Range, HTTPOS uses the method based on TCP MSS + TCP advertising window (Section 4.2.2). Otherwise, HTTPOS selects the method based on multiple TCP connection + HTTP Range (Section 4.2.3) if the server does not support HTTP Pipelining, or the method based on HTTP

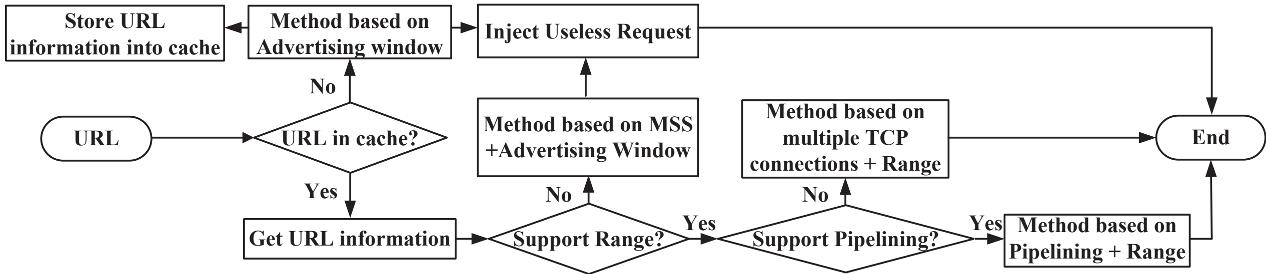


Figure 3: The HTTPOS operations.

Pipelining + HTTP Range (Section 4.2.4) if the server does.

Since TCP-based methods (i.e., methods based on TCP MSS and TCP advertising window) cannot change the size of web objects, HTTPOS injects a useless HTTP request between two requests as described in Section 3.3.2 to mislead the attacks that exploit the characteristics of web objects. It is worth noting that if an attacker cannot observe the TCP header, HTTPOS can also use retransmitted packets to mislead those attacks. Table 3 summarizes all methods in HTTPOS and the corresponding attacks that can be evaded.

4.2 Modules

4.2.1 Method based on TCP advertising window

Since a TCP sender cannot send more data than the advertising window permits, HTTPOS controls the size of incoming response packets by manipulating the advertising window in each outgoing packet. More precisely, given a web object of S bytes, HTTPOS selects N_v integers $\{v_1, v_2, \ldots, v_{N_v}\}$, where v_i is the advertising window in the ith outgoing TCP packet and $\sum_{i=1}^{N_v} v_i = S$. HTTPOS sends a new TCP packet only after receiving a TCP data packet from the server to prevent the server from combining the advertising windows in several TCP packets and then sending a large packet that may be recognized by a traffic analysis.

4.2.2 Method based on TCP MSS + TCP advertising window

Since the method based on TCP advertising window allows only one TCP packet to be sent in an RTT, it may introduce large delay. To address this problem, we propose a new method that employs both TCP MSS and TCP advertising window. This method is motivated by two observations. First, given a web object of S bytes and a default MSS of M bytes, a successful traffic-analysis attack usually relies on the last packet whose size is equal to $S \mod M$. Second, a TCP sender can send several M-byte TCP data packets in an RTT. HTTPOS therefore can either change the default MSS or manipulate the last packet's size by using TCP advertising window. In the former case, after setting the MSS to M', the last packet's size becomes $S \mod M'$. In the latter case, HTTPOS sets the advertising window to $\lfloor \frac{S}{M} \rfloor M$ in order to fetch the first $S - R$ bytes, where $R = S \mod M$. After that, it sets the advertising window to $R - r$ bytes, where r is a random positive integer less than R, to download $R - r$ bytes and then announces an advertising window larger than r to get the remaining r bytes. Therefore, if the normal operation needs one RTT to download the R bytes, HTTPOS may use an additional RTT to download it (i.e., one RTT for $R - r$ bytes and another RTT for r bytes). If the server's TCP stack increases its congestion window based on the number of valid ACK packets, HTTPOS could send customized ACK packets to induce the server to increase its congestion window quickly.

4.2.3 Method based on multiple TCP connections + HTTP Range

If a server supports HTTP Range but does not support HTTP Pipelining, HTTPOS establishes multiple TCP connections and sends partial `GET` requests for a web object in parallel to the server. As explained in Section 3.3.1, HTTP Range can limit the size of response packets. The server will process these requests simultaneously if the server adopts multi-threading and then return the web object through several packets. HTTPOS will re-organize the responses before delivering the content to the client.

4.2.4 Method based on HTTP Pipelining + HTTP Range

If a server supports both HTTP Pipelining and HTTP Range, HTTPOS puts several partial `GET` requests into one packet and sends it out. The server will process these requests one by one and send back the responses. Without the need to wait for the arrival of a response before sending another partial `GET` request, the additional delay introduced by HTTPOS will decrease.

Table 3: Methods in HTTPOS and the corresponding attacks that can be evaded.

Methods	Layers	SSWRPQ	BLJL	LL-JC	LL-NBC	CWWZ
Method based on Advertising Window	TCP		√	√	√	√
Method based on MSS + Advertising Window	TCP		√	√	√	√
Method based on Multiple TCP connections + HTTP Range	HTTP	√	√	√	√	√
Method based on HTTP Pipelining + HTTP Range	HTTP	√	√	√	√	√
Inject Useless Request	HTTP	√	√			√
Inject Packet Delay	TCP		√			

4.3 Implementations

We implemented HTTPOS in C with 3022 lines of code (reported by *CLOC* [31]) and tested it on Ubuntu 9.04 with 2.6.27 kernel. To manipulate TCP packets, HTTPOS uses `iptables` (version 1.4.0) and the `libnetfilter_queue` library (version 0.0.16) to hook outgoing TCP packets of interest. HTTPOS adds rules into `iptables`' `INPUT` and `OUTPUT` chains, so that the packets matching the rules will be queued in the kernel. HTTPOS acquires a packet through the `libnetfilter_queue` library and then modifies it (e.g., the advertising window and the MSS option) before releasing it. Additional delay is introduced to the outgoing packets if needed. HTTPOS uses raw socket to inject TCP packets if necessary and employs the `libpcap` 1.0.0 library to capture TCP packets for verification. Moreover, the POSIX Threads (`pthreads`) library was utilized to create and manage multiple threads for multiple HTTP/TCP connections between clients and HTTPOS, and those between HTTPOS and web servers.

In our measurement experiments to be discussed in Section 5.1, we established an IPSec tunnel as an example of IP tunnel, built an SSH tunnel as an example of TCP tunnel, and set up a wireless channel encrypted by WPA. HTTPOS uses different modules to handle HTTP requests and responses for different scenarios. When the IPSec tunnel and the wireless channel are used, HTTPOS acts as an HTTP proxy. When the SSH tunnel is employed, HTTPOS behaves as a SOCKS proxy for users to visit the Internet. At the same time, HTTPOS communicates with the SSH tunnel through SOCKS 4 [25], because the SSH port forwarding provides service via SOCKS. For the HTTPS channel, HTTPOS is implemented as a Firefox add-on to manipulate HTTP requests before they are sent to the SSL/TLS layer. In all scenarios, HTTPOS can modify the header of HTTP requests and insert useless requests if necessary.

We also implemented those traffic-analysis attacks introduced in Section 3.1 using Python and Weka 3.6.1 [36] to evaluate the effectiveness of HTTPOS. In Section 5, we report their accuracy with and without HTTPOS.

4.4 Measuring the support rates of the TCP and HTTP based control

HTTPOS exploits the basic protocol features in TCP and HTTP described in RFC 793 and RFC 2616, respectively. Since not all servers comply with the RFCs, we conducted two sets of measurement to evaluate whether operating systems and web servers support manipulating packet size through TCP MSS, TCP advertising window, HTTP Pipelining, and HTTP Range. In the first set, we tested popular operating systems and web servers with their default settings in our test-bed. In particular, we tested Apache v2.3.6, nignx v0.8.42 and lighttpd v1.4.26 in a Ubuntu machine (kernel 2.6.28) and IIS v7.5 in a Windows 7 box. We selected these web servers, because they represent more than 90% market share [30]. Since the Google web server cannot be downloaded, we cannot test it in the test-bed.

In the second set, we targeted on the top 2000 web sites in the Alexa rankings [1]. We modified `Pagestats` [16] to drive Firefox 3.6.3 to automatically visit these web sites. Since Firefox downloads all the necessary web objects, which may be located in different web servers, we managed to collect 143,333 URLs in 8,845 web servers after Firefox visited the front pages of the 2000 web sites. We used NetCraft's service [5] to identify the operating system and the web server software used by each server. Since NetCraft resolved the web server software used in only 5884 web servers, we employed `httprecon-7.3` [32] to further infer the web server software in the remaining 2961 servers. There are still 1181 servers whose web server software cannot be identified by `httprecon-7.3`, and we refer them to as "others." Moreover, since NetCraft identified 4957 web servers' operating systems, we group the other 3888 servers as "others." For the Google web server which has different names [4], we crawled 4622 URLs starting from `http://www.google.com.hk/intl/zh-TW/options/` and extracted the names of the web server software from the `Server` field in the response header. As a result, we obtained a total of 231 Google servers.

Table 4: **Major operating systems' support rate of TCP MSS negotiation and TCP advertising window based control.**

OSes	ADV_L = 2000 bytes			MSS_L = 1460 bytes (the default)			$ADV_L = MSS_L$ bytes		
(No. of servers)	MSS_L=128	MSS_L=256	MSS_L=536	ADV_L=128	ADV_L=256	ADV_L=536	MSS_L=128	MSS_L=256	MSS_L=536
Windows (388)	88.40%	89.43%	100.00%	95.36%	95.36%	97.42%	99.22%	99.48%	100.00%
Linux (3875)	97.90%	98.63%	100.00%	99.17%	99.32%	99.50%	99.76%	99.94%	100.00%
AIX (19)	84.21%	100.00%	100.00%	94.73%	94.73%	94.73%	100.00%	100.00%	100.00%
Solaris (71)	98.59%	100.00%	100.00%	97.18%	97.18%	98.59%	100.00%	100.00%	100.00%
FreeBSD (224)	25.89%	99.55%	99.55%	99.10%	99.10%	99.10%	99.10%	100.00%	100.00%
BIG-IP (380)	98.68%	99.21%	100.00%	99.47%	99.47%	99.47%	99.73%	100.00%	100.00%
Others (3888)	84.90%	96.38%	99.89%	96.94%	97.38%	97.94%	99.61%	99.76%	99.94%

4.4.1 TCP MSS and TCP advertising window

To test whether a server allows HTTPOS to manipulate the packet size through TCP MSS, we modify the advertised MSS values in the TCP option. Let MSS_L be the MSS value announced by us in the TCP SYN packet and MSS_R be the MSS value returned in the server's TCP SYN/ACK packet. We let MSS_L be less than the typical value for MSS_R (which is 1460 bytes in most cases). Moreover, MSS_L should never appear in the flow between the client and web server. If indeed $MSS_R > MSS_L$, we send an HTTP request to download a web object larger than MSS_R. If the payload sizes of all response packets are less than or equal to MSS_L, then the server permits HTTPOS to control its packet size.

To test whether a server allows HTTPOS to control the size of response packet through TCP advertising window, we first disable TCP window scale option and then change the advertising window, denoted as ADV_L, of an outgoing TCP packet to a value smaller than MSS. Similar to the previous case, if the response packet's payload size is less than or equal to ADV_L, then the server allows HTTPOS to control its packet size.

Since TCP MSS and TCP advertising window can be set to arbitrary values, we could not enumerate all possible combinations. Instead, we investigated three scenarios: (1) $ADV_L = 2000$ bytes and $MSS_L = \{128, 256, 536\}$ bytes; (2) use the default MSS announced by the remote server (i.e., $MSS_L = MSS_R = 1460$ bytes) and $ADV_L = \{128, 256, 536\}$ bytes; and (3) $MSS_L = ADV_L = \{128, 256, 536\}$ bytes. Table 4 summarizes the measurement results. Under most settings, more than 85% servers allow HTTPOS to control their packet size. In particular, the support rate increases when either the MSS or the advertising window increases. Moreover, when the MSS and advertising window use the same value, most servers support HTTPOS. For example, for $MSS_L = ADV_L = 536$ bytes, 8843 out of 8845 servers allow HTTPOS to control their packet size.

For $MSS_L = 128$ bytes and $ADV_L = 2000$ bytes, only 25.89% of the FreeBSD servers allow HTTPOS to control their packet size. The reason is that FreeBSD sets its default minimal MSS to 216 bytes to prevent TCP MSS resource exhaustion attacks [2]. However, this low support rate does *not* mean that HTTPOS cannot evade those traffic-analysis attacks for FreeBSD servers. First, HTTPOS can still control the packet size by setting $MSS_L = ADV_L = 128$ bytes, which has 99.10% support rate. Second, Figure 4 shows that the HTTP headers in more than 90% of the responses from our data sets are larger than 256 bytes. Therefore, it is sufficient for HTTPOS to use $MSS_L = 256$ bytes for which the support rate for FreeBSD is 99.55%. Finally, even though HTTPOS cannot force the response packets to be 128 bytes or less, the actual payload size already differs from the one when default MSS_L is used. As a result, the new payload size also helps a user evade the traffic-analysis attacks.

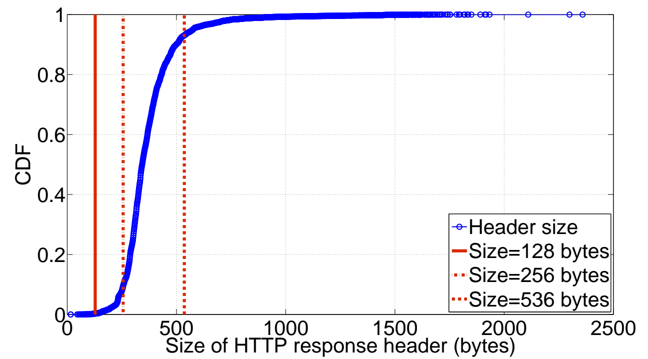


Figure 4: CDF of the size of HTTP response headers based on our data sets of $143,333$ **URL responses obtained from** 8845 **web servers.**

4.4.2 HTTP Range and HTTP Pipelining

We discover that some web applications may ignore HTTP Range requests even if the underlying web server supports HTTP Range. Therefore, we measure the support rate of HTTP Range on both the URL level and the web server level. To test whether a URL supports HTTP Range, we send partial `GET` requests to the server and then inspect the response of `Accept-Ranges`. If the server replies

Table 5: **The support rates of HTTP Range and HTTP Pipelining in terms of the number of servers.**

Web servers (No. of servers)	HTTP Range	HTTP Pipelining	HTTP Range+Pipelining
Apache (4249)	84.00%	63.90%	58.80%
IIS (1738)	76.06%	77.00%	65.88%
nginx (1103)	80.15%	75.16%	70.35%
lighttpd (367)	84.47%	74.70%	68.94%
Others (1388)	73.34%	65.13%	55.55%

Table 6: The support rates of HTTP Range and HTTP Pipelining in terms of the number of URLs.

Web servers (No. of URLs)	HTTP Range	HTTP Pipelining	HTTP Range+Pipelining
Apache (59698)	89.02%	79.71%	68.80%
IIS (22485)	85.03%	88.24%	73.38%
nginx (18714)	83.16%	87.58%	70.74%
lighttpd (5506)	82.64%	84.87%	67.51%
Others (36930)	66.74%	69.31%	53.98%

Table 7: The Google web servers' support rates of HTTP Range and HTTP Pipelining in terms of the number of servers.

Google web servers (No. of servers)	HTTP Range	HTTP Pipelining	HTTP Range+Pipelining
sffe (38)	100%	100%	100%
DFE/largefile (109)	100%	100%	100%
GSE (24)	58.33%	100%	58.33%
codesite (2)	0%	100%	0%
Others (58)	0%	100%	0%

Table 8: The Google web servers' support rates of HTTP Range and HTTP Pipelining in terms of the number of URLs.

The Google web servers (No. of URLs)	HTTP Range	HTTP Pipelining	HTTP Range+Pipelining
sffe (2580)	99.88%	100%	99.88%
DFE/largefile (906)	100%	100%	100%
GSE (461)	48.59%	100%	48.59%
codesite (335)	0%	100%	0%
Others (340)	0%	100%	0%

with "`Accept-Ranges: bytes`," it supports HTTP Range; otherwise, it may send back "`Accept-Ranges: none`" or nothing. On the web server level, we regard a server as supporting HTTP Range if one URL on that server supports HTTP Range.

We also discover that if a web server supports HTTP Pipelining, all web applications running on that web server also support HTTP Pipelining. To test whether a server supports HTTP Pipelining, we send out several HTTP requests together, each of which carries "`Connection: keep-alive`," without waiting for the corresponding responses. If the server responds to all these requests, it is considered supporting HTTP Pipelining. Otherwise, the server may just respond to the first request and then close the connection.

According to our first set of experiments, all those major web servers with default settings support both HTTP Range and HTTP Pipelining. Besides, Tables 5 and 6 show the measured support rates of the HTTP features from $143,333$ URLs located in 8845 servers. In particular, we find that $117,025$ URLs (81.6%) from 7103 servers (80.3%) support HTTP Range, $114,087$ URLs (79.6%) from 6060 servers (68.5%) support HTTP Pipelining, and $94,458$ URLs (65.9%) from 5545 servers (62.7%) support both HTTP Range and HTTP Pipelining.

Tables 7 and 8 show the Google web servers' support rates of HTTP Range and HTTP Pipelining in terms of the number of servers and URLs, respectively. We find that all the Google web servers support HTTP Pipelining, but only "sffe," "DFE/largefile," and a partial of "GSE" support HTTP Range.

5 Evaluation

In this section, we present the results of evaluating HTTPOS in terms of its effectiveness on defeating the traffic-analysis attacks and its impact on the goodput of fetching web objects.

5.1 Experiment settings

We first downloaded the front pages from the top 100 web sites ranked by Alexa [1]. For web sites that belong to the same company and have similar web page layouts, we tested only the site having the highest rank. For example, Google owns several sites having high ranks (such as `google.com`, `google.com.hk`, and `google.de`), and we just tested `google.com`. Moreover, we replaced porn sites with other top web sites. We used Firefox 3.6.3 equipped with Flash plugin 10 to visit the web sites. To automate the experiments, we prepared a Python script to invoke modified `Pagestats` [16] to visit each web site and used TCPDump to capture the trace. We refer to the process of visiting all the web sites once as a *round*, and we performed a total of 100 rounds of measurement experiment. The traces in odd-numbered rounds were used to train classification algorithms, and the trained models were tested on traces in even-numbered rounds. Based on `Pagestats`'s results, we computed the goodput as the ratio of total bytes fetched to the download time.

We examined two deployment scenarios for HTTPOS: on the browser side when IP tunnel, encrypted wireless channel, or HTTPS channel is used and at a TCP tunnel's entry point. To establish an IP tunnel, we employed L2TP v1.2.0 and OpenSwan v2.6.24 to build an IPSec tunnel between two endpoints. To set up the wireless channel, we used a laptop with Intel PRO/Wireless 2200BG Mini-PCI Adapter to connect to an Access Point with WPA1 encryption enabled in our laboratory and employed AirPcap [12] to capture wireless frames. For TCP tunnels, we used SSH port forwarding to create a TCP-based tunnel between two endpoints following the configuration in [26].

It is important to point out that our experiment settings are actually favorable to an attacker for the following reasons.

1. Koukis et al. [24] showed that parsing mixed web sessions in packet traces obtained from an encrypted tunnel is very difficult. However, in our case we saved all the packets belonging to a web session into a separate `pcap` file, thus removing this obstacle for the attacker.

2. Coulls et al. [14] pointed out that a web browser's caching may significantly affect the accuracy of an attack, because the browser does not need to download web objects in the cache, thus affecting the flow size. This problem, however, does not occur to our case, because `Pagestats` [16] always clears Firefox's cache after visiting a web site/page.

3. Liberatore et al. [26] reported that a large delay between the training data set and the test data set may cause lower accuracy. In particular, they observed a decrease from 73% to 63% for a delay of four weeks. The delay in our case, however, is small (i.e., around 30 mins), and the 100 rounds of experiments were carried out continuously.

4. By using the traces from every other round of measurement, we provide the attacker with a much more accurate view of the traffic. In a realistic attack scenario, an attacker normally spends some time to learn from the captured traffic. Therefore, the traffic pattern may not be the same as those she has observed before when the attack is finally launched.

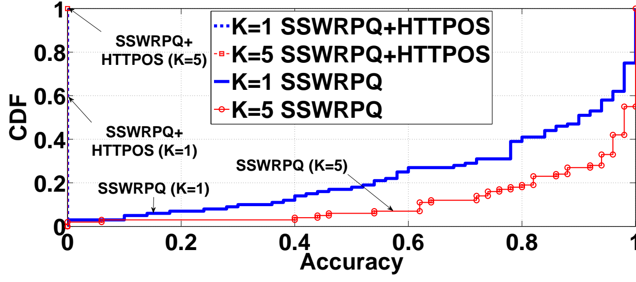
5.2 Evasion evaluation

5.2.1 Defeating the SSWRPQ, BLJL, LL-JC and LL-NBC attacks

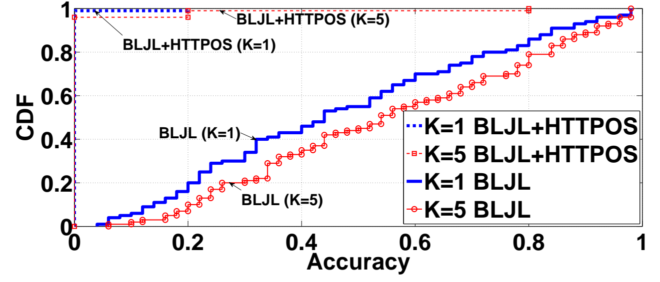
To evaluate the effectiveness of HTTPOS against the four attacks targeting on identifying web sites (i.e., the SSWRPQ, BLJL, LL-JC, and LL-NBC attacks), our approach is to compare the attack accuracy with and without applying HTTPOS to the encrypted traffic. To compute the attack accuracy for a given web site, we first compute the similarity between the trace obtained for the web site and the available profiles based on the attack methods introduced in Section 3.1. For each attack, we then sort the similarity and select the top K web sites as our inference. The attack is considered successful if the actual web site is one of the K web sites selected by the attack. Clearly, the likelihood of making a correct decision increases with K. The attack accuracy for the web site is then given by the percentage of successful attacks obtained from the 50 rounds of measurement.

Figure 5 shows the CDF of the attack accuracy for the four attacks with and without applying HTTPOS to the traffic flowing through an IPSec tunnel. Similarly, Figure 6 reports their accuracy for SSH tunnel. The two solid curves show the attack accuracy without HTTPOS, whereas the two dashed curves are the results when HTTPOS is used. The figures show that without using HTTPOS the attacks can identify the visited web sites with high fidelity. For $K = 1$, the LL-JC attack on IPSec traffic achieves at least 70% accuracy for guessing the 100 web sites, where around 70% of the web sites are correctly identified from *all* 50 rounds (i.e., 100% accuracy). The LL-NBC attack achieves similar performance, where at least 80% accuracy is achieved for each site, and more than 60% of the web sites are identified with 100% accuracy. When targeting on SSH traffic, the LL-JC (LL-NBC) attack achieves 100% accuracy for more than 75% (90%) of the web sites with more than 60% (90%) accuracy for each site. We also observe that the LL-JC and LL-NBC attacks have better performance than the SSWRPQ and BLJL attacks, and all the four attacks achieve a better accuracy for $K = 5$.

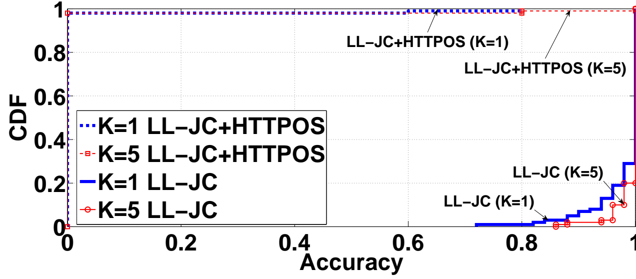
With HTTPOS, the accuracy of the four attacks drop significantly. Figures 5 and 6 show that, for $K = 1$, *none* of the attacks can achieve 100% accuracy for any web site. For IPSec traffic, the accuracy of these attacks drops to 0% for at least 98% of the web sites, because they fail to make a single correct decision for the majority of those web sites. For $K = 5$, the SSWRPQ, BLJL, LL-JC, and LL-NBC attacks still suffer from 0% accuracy for 100%, 98%, 96%, and 94% of the web sites, respectively. The "better" performance achieved by the LL-NBC attack is possibly due to the KDE which considers some packet sizes that never appear but are close to the sizes in the training data set. We also observe similar (poor) performance for SSH traffic, for which these attacks achieve 0% accuracy for at least 98% (95%) of the web sites for $K = 1$ ($K = 5$).



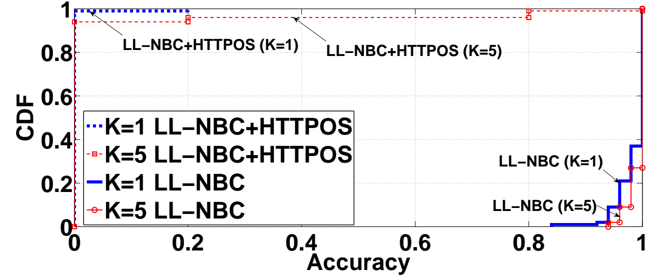
(a) The SSWRPQ attack.



(b) The BLJL attack.

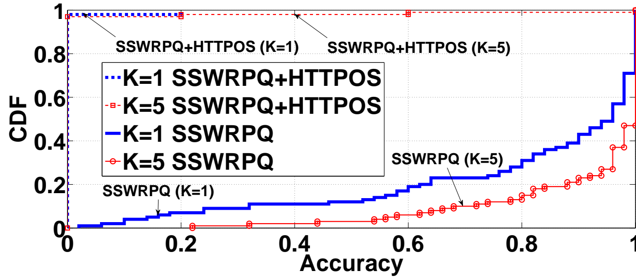


(c) The LL-JC attack.

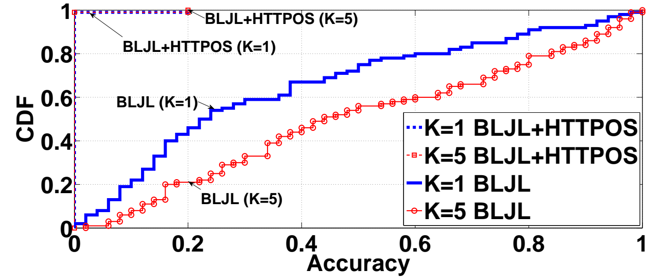


(d) The LL-NBC attack.

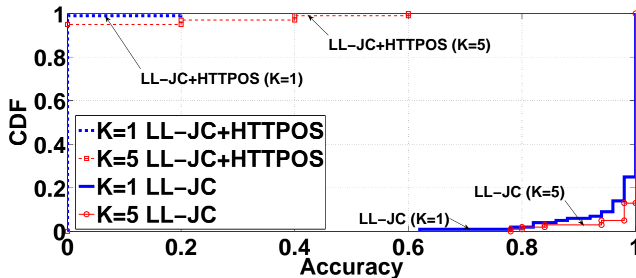
Figure 5: Attack accuracy for the SSWRPQ, BLJL, LL-JC, and LL-NBC attacks with and without applying HTTPOS to the traffic in an IPSec tunnel.



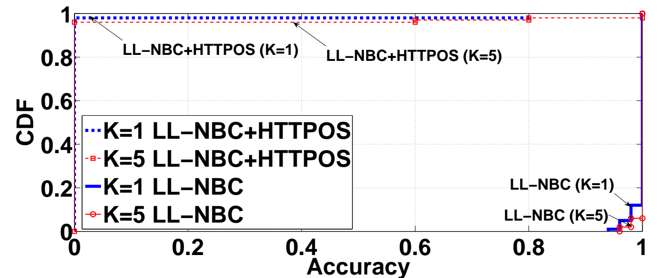
(a) The SSWRPQ attack.



(b) The BLJL attack.



(c) The LL-JC attack.



(d) The LL-NBC attack.

Figure 6: Attack accuracy for the SSWRPQ, BLJL, LL-JC, and LL-NBC attacks with and without applying HTTPOS to the traffic in a SSH tunnel.

5.2.2 Evading the CWWZ attack

We use only the Google search engine to evaluate HTTPOS against the CWWZ attack for two main reasons. The first is that the search engine example in Section IV.D of [13] provides all the key features of the CWWZ attack. The second is that the actual URLs for the web sites (i.e., OnlineHealth, OnlineTax and OnlineBank) reported in Chen et al.'s paper [13] have not been revealed. Since the Google search engine provides both HTTP and HTTPS services, we conducted

experiments for both services to evaluate HTTPOS's effectiveness. Since the Google search engine runs on the GWS web server and does not support HTTP Range, HTTPOS uses the TCP-based methods and injects useless requests to evade the CWWZ attack.

We first consider the scenario of communicating with the Google search engine through an encrypted wireless channel. The experiment setting is the same as the one in Section IV.D of [13]. Before showing the experiment results with various inputs, we use an example to illustrate how HTTPOS can evade the CWWZ attack. In this example, the values in a vector sequence are the payload sizes of wireless frames. We entered the word "hi" in the Google search input box and the vector sequence of directional packet sizes [13] is ($556 \rightarrow$, $451 \leftarrow, 557 \rightarrow, 463 \leftarrow$), where the request packets carrying "h" and "hi" are of 556 bytes and 557 bytes, respectively, and the corresponding response packets are of 451 bytes and 463 bytes. If HTTPOS pads the HTTP request headers with useless fields to the same size, sets $MSS_L = 200$ bytes, and disables the TCP window-scaling bit, then we get a new vector sequence ($572 \rightarrow$, $260 \leftarrow$, $260 \leftarrow$, $75 \leftarrow$, $572 \rightarrow$, $260 \leftarrow$, $260 \leftarrow$, $87 \leftarrow$). Since the CWWZ attack cannot not infer the user input from this sequence, its reduction power is dampened to one.

However, a smart attacker might group several packets together and rebuild a correct vector sequence. For example, by subtracting 72 bytes—the payload size of a wireless frame carrying a TCP ACK packet—from the size of the first response packet in the original vector sequence, an attacker may infer the size of TCP payload as $451 - 72 = 379$ bytes. Similarly, by subtracting 72 bytes from the sizes of the first three response packets in the new vector sequence and then summing the remainder, the attacker obtains $260 - 72 + 260 - 72 + 75 - 72 = 451 - 72 = 379$ bytes, therefore recovering the original vector sequence. To defeat this attack, we inject a useless request between two requests as described in Section 3.3.2 and therefore obtain another vector sequence ($572 \rightarrow$, $260 \leftarrow$, $572 \rightarrow$, $260 \leftarrow$, $75 \leftarrow$, $260 \leftarrow$, $260 \leftarrow$, $82 \leftarrow$, $572 \rightarrow$, $260 \leftarrow$, $260 \leftarrow$, $572 \rightarrow$, $87 \leftarrow$, $260 \leftarrow$, $260 \leftarrow$, $103 \leftarrow$). Since HTTPOS can inject various requests from the ambiguity set, the attacker cannot restore the original vector sequence from this new sequence.

Yet an even more advanced attacker may still be able to infer a set of keywords sent by the user from some special packet sizes (e.g., 75 bytes and 87 bytes in the above example). We propose the following method to address this challenge. Since the Google server supports HTTP Pipelining, HTTPOS sends out the request for a user input and a useless request in one or more successive packets with a *zero* advertising window. The server will process both requests and store the responses in its TCP/IP stack, because the zero advertising window forbids it to send back the responses immediately. After a short period (e.g., 30 ms according to our evaluation with the Google server), HTTPOS sends an ACK packet with a large advertising window (e.g., 2000 bytes), and the server is induced to pack the responses into blocks of MSS-byte packets. For the above example, the vector sequence becomes ($1072 \rightarrow$, $837 \leftarrow$, $1072 \rightarrow$, $870 \leftarrow$), which is completely different from the original one.

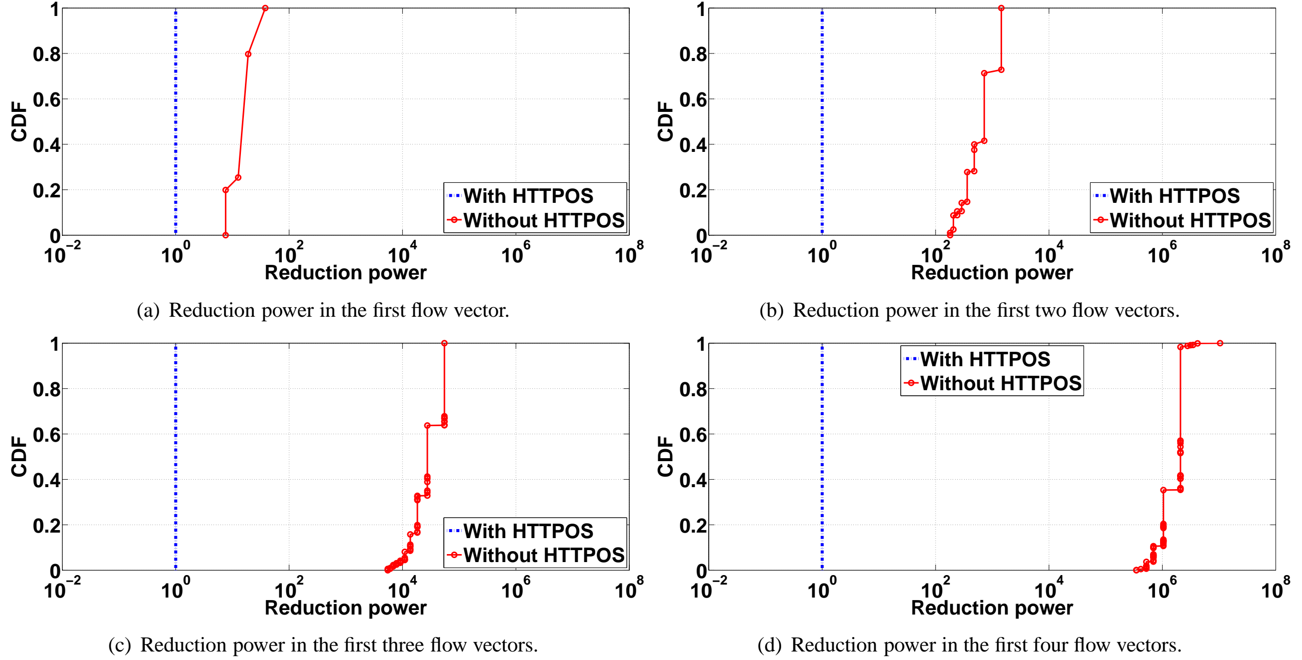
We also evaluated HTTPOS using the Google HTTPS-based search service (i.e., `https://www.google.com`). In this setting, the values in a vector sequence are the sizes of the TCP packet payload. When we entered "hi" in the search input box, we observed a vector sequence ($539 \pm 20 \rightarrow$, $679 \leftarrow$, $540 \pm 20 \rightarrow$, $658 \leftarrow$)[1]. With HTTPOS, the vector sequence becomes ($1065 \rightarrow, 1323 \leftarrow$, $1065 \rightarrow$, $1304 \leftarrow$). Consequently, the attacker can neither find a similar vector from her training data set nor recover the original vector, thus diminishing the CWWZ attack's reduction power to one.

Figures 7 and 8 plot the CDFs of the CWWZ attack's reduction power with and without HTTPOS. A larger reduction power indicates that the CWWZ attack has a stronger capability to infer the visited web pages. Note that the minimal reduction power is one, meaning that the CWWZ attack cannot infer any useful information from each observed flow vector. In this experiment, we follow the steps in [13] and randomly choose 1000 popular search key words from Google Trend [3]. Since these key words contain only characters $\{a, b, ..., z, 0, 1, ..., 9, dot, space\}$, the size of the ambiguity set is $k = 38$.

Figures 7(a)-7(d) plot the results when the CWWZ attack is applied to the HTTP traffic between HTTPOS and the Google search engine through an encrypted wireless channel. We only consider the first four characters (i.e., $n \leq 4$), because they are sufficient for showing HTTPOS's effectiveness. That is, only the first four flow vectors are examined. Obviously, without HTTPOS, the CWWZ attack achieves large reduction power as n increases. The attacker can then determine the words sent to the search engine. However, with HTTPOS, the reduction power is fixed to one and does not change along with n. Therefore, the attacker cannot gain any information from the observed flow.

Figures 8(a)-8(d) plot the results for the HTTPS traffic. Without HTTPOS, the reduction power increases with n. When n reaches 4, more than 50% of the key words have the reduction power around 10^6, which can reduce the huge original ambiguity set (whose size is 38^4) to a small set (whose size is $38^4/10^6 \simeq 2$). However, even for $n = 4$, HTTPOS can force the reduction power for all the 1000 key words to one. In other words, the attacker must guess the key word based on the ambiguity set with 38^4 possibilities.

[1]The parameter "gs_gbg" in the queries to the Google HTTPS-based search engine introduces ± 20 random bytes to each request.



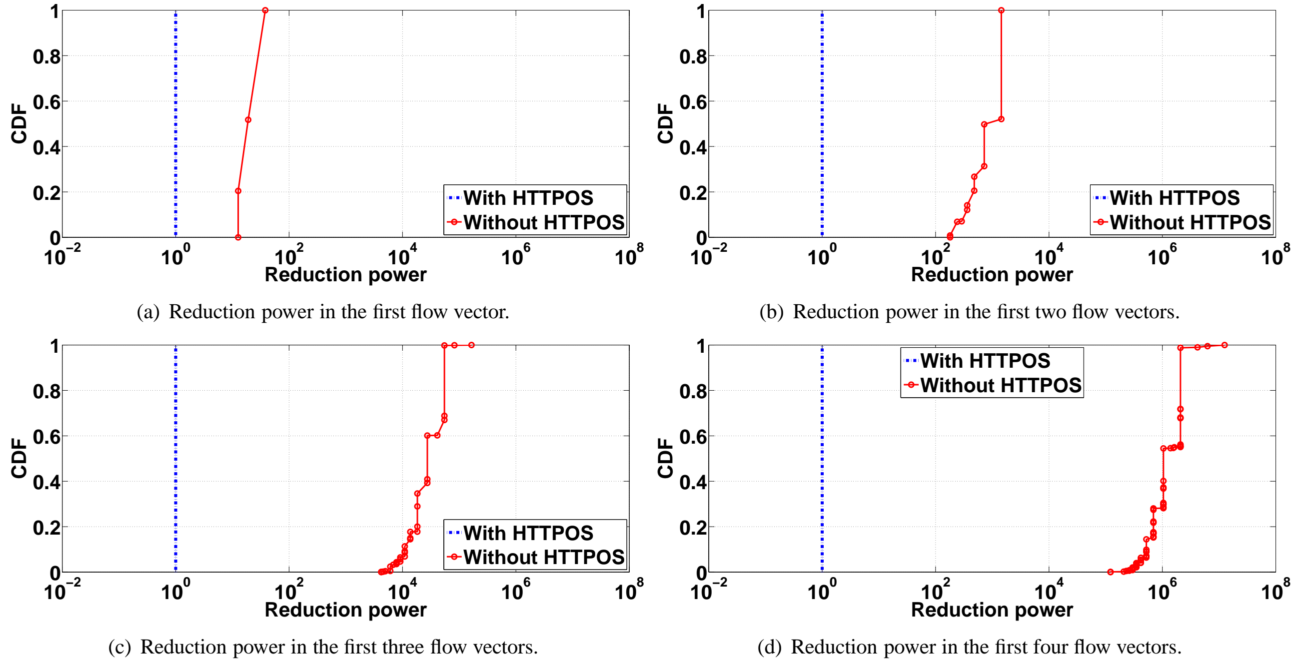
(a) Reduction power in the first flow vector.

(b) Reduction power in the first two flow vectors.

(c) Reduction power in the first three flow vectors.

(d) Reduction power in the first four flow vectors.

Figure 7: Reduction power of the CWWZ attack on the HTTP traffic between HTTPOS and the Google search engine through an encrypted wireless channel.



(a) Reduction power in the first flow vector.

(b) Reduction power in the first two flow vectors.

(c) Reduction power in the first three flow vectors.

(d) Reduction power in the first four flow vectors.

Figure 8: Reduction power of the CWWZ attack on the HTTPS traffic between HTTPOS and the Google search engine.

5.3 Performance evaluation

5.3.1 Evaluation of individual methods

To evaluate the effect of each HTTPOS method on the performance, we randomly selected 1000 URLs that support all methods. We set $ADV_L = 200$ bytes for the method based on TCP advertising window, let $MSS_L = 200$ bytes for the method based on TCP MSS, and used three TCP connections for the multiple connections method. Moreover,

we splitted each web object into three parts for the HTTP Range method. Note that the following results do not represent the best performance that HTTPOS can achieve.

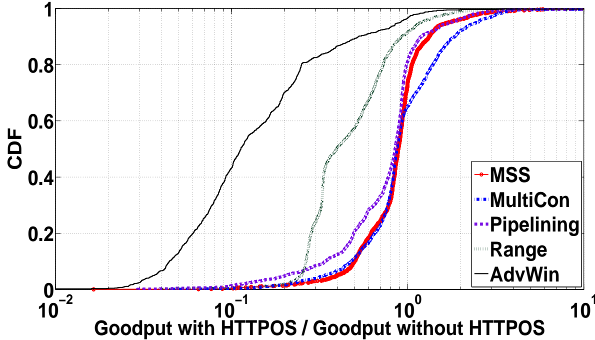


Figure 9: The ratio of the resultant goodput for each HTTPOS method to that without HTTPOS.

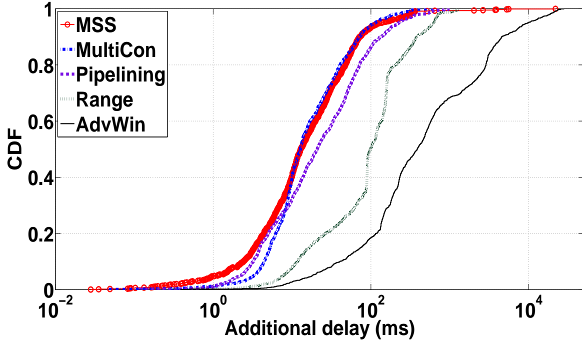


Figure 10: Additional delay introduced by each HTTPOS method.

Figure 9 plots the ratio of the resultant goodput for each HTTPOS method to that without using HTTPOS. For brevity, we use *MSS* to denote the method based on TCP MSS + TCP advertising Window, *MultiCon* the method based on Multiple TCP Connections + HTTP Range, *Pipelining* the method based on HTTP Pipelining + HTTP Range, *Range* the method based on pure HTTP Range, and *AdvWin* the method based on TCP advertising window. As shown, HTTPOS can achieve at least 80% goodput for 75% of the URLs when the MSS method or the MultiCon method is applied, and 90% goodput for 50% of the URLs when the Pipelining method is applied.

The performance degradation introduced by the MSS method is not significant, because, as discussed in Section 4.2.2, it only needs an additional RTT to finish the transmission. On the other hand, we notice that unlike an IP tunnel, a TCP tunnel may multiplex multiple TCP connections into a single TCP tunnel which could become the performance bottleneck. To tackle this problem, we establish several TCP tunnels in advance and divert TCP connections into different TCP tunnels to achieve parallel transmissions. Moreover, as expected, the AdvWin method gives the worst performance, because it allows only a single packet transmission from the server in an RTT. Moreover, we use a very small ADV_L (i.e., 200 bytes) for this evaluation.

Although some methods may cause certain URLs to experience a low goodput, we find that the additional delay introduced by these methods is not significant under our parameter settings. Figure 10 reveals the additional delay introduced by each HTTPOS method. As shown, the MSS, MultiCon, and Pipelining methods introduce less than 100 ms delay for more than 90% of the URLs. The Range method, on the other hand, introduces less than 200 ms delay for more than 80% of the URLs.

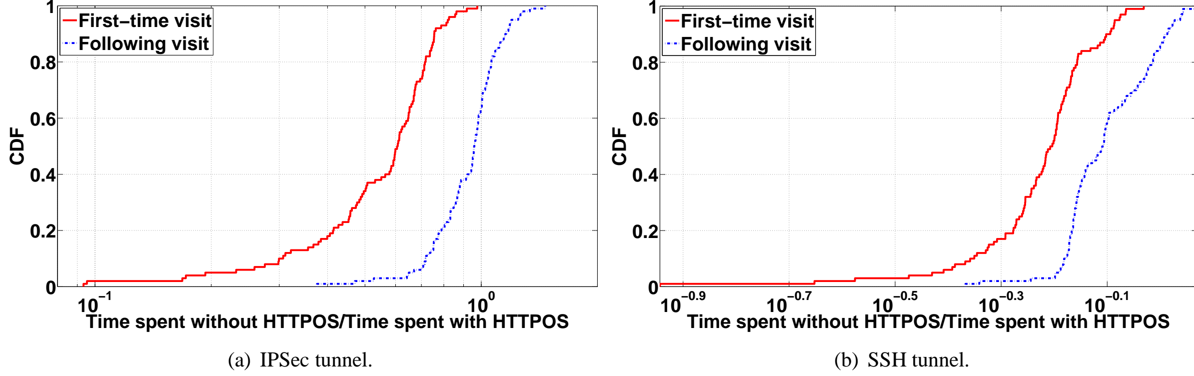
5.3.2 Impacts on the performance of Internet browsing

To evaluate the overall performance of HTTPOS, we visited each of the top 100 web sites 10 times with and without applying the HTTPOS operations depicted in Figure 3. We recorded the download time based on the output of `Pagestats` for each site. Figure 11(a) and Figure 11(b) show the CDFs of the ratio for the download time without and with HTTPOS when using IPSec tunnel and SSH tunnel, respectively. The figures show clearly that the first-time visit to each site via HTTPOS needs more time than the normal visits (i.e., all values less than one), because HTTPOS uses the method based on advertising window. When HTTPOS is applied, the time for visiting 60 sites through IPSec is at most 1.6 times of the time without HTTPOS.

However, once the URL information is cached, the time required for the following visits is close to the time for the normal visits (i.e., value close to one). The time for visiting 60 sites through IPSec is at most 1.1 times of the time without HTTPOS. Furthermore, the time for visiting more than 90 sites through IPSec is at most 1.4 times of the time without HTTPOS. The reason is that for each URL, HTTPOS will select the method with the least impact on the performance while not compromising the protection capability according to Figure 3. It is also interesting to note that as a result of employing multiple TCP connections, HTTPOS may even enjoy better performance than the normal visits (i.e., values larger than one) in some cases. As shown in Figure 11(a), visiting around 40 out of the 100 sites through IPSec requires less time than the normal visits.

5.3.3 Impacts on the performance of Google search

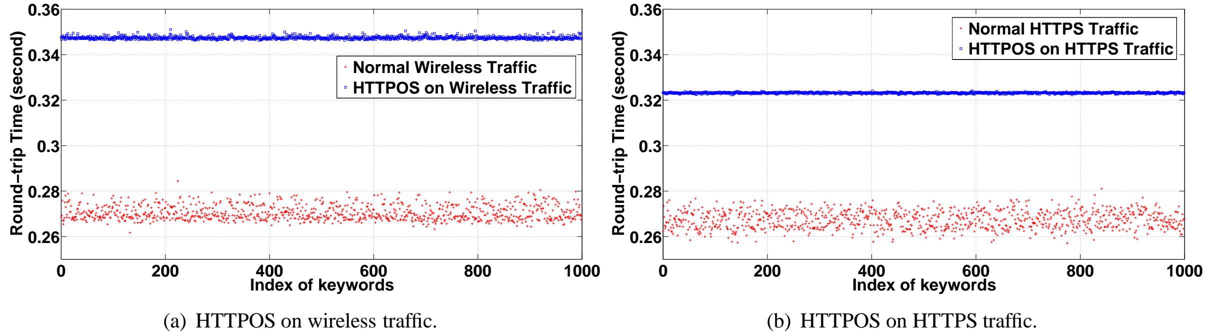
To evaluate the impacts of HTTPOS on the performance of using Google search, we measured the RTT from the epoch when the user sends a query to the epoch when the user receives the response with and without HTTPOS. Figure 12(a) illustrates the RTTs obtained from the scenario where a user in Hong Kong visited the Google search service (i.e.,



(a) IPSec tunnel.

(b) SSH tunnel.

Figure 11: The effect of HTTPOS on the performance of Internet browsing.



(a) HTTPOS on wireless traffic.

(b) HTTPOS on HTTPS traffic.

Figure 12: The effect of HTTPOS on the performance of using Google search.

74.125.47.147) through a 802.11g wireless link. Figure 12(b) shows the RTTs obtained from the scenario where the user accessed the same search service through HTTPS. In both experiments, the user entered 1000 popular search keywords from Google Trend [3] for 30 times, and the respective median RTTs for each keyword is shown in Figure 12(a) and Figure 12(b).

In this experiment, HTTPOS employs the technique described in Section 5.2.2 to evade the CWWZ attack. More precisely, HTTPOS puts the real request and a useless request in one packet, and sets $ADV_L = 0$ before dispatching the packet to the server. After a small delay, HTTPOS sends an ACK packet to announce a large advertising window and induce the server to send back responses. The delay is 120 ms for wireless traffic and 100 ms for HTTPS traffic, respectively. Figure 12 shows that the additional delay introduced by HTTPOS is small, because the server can send back the responses immediately upon receiving the ACK packet. The additional delay is less than 80 ms in Figure 12(a) and less than 60 ms in Figure 12(b).

5.4 Discussion

We believe that HTTPOS significantly raises the bar and makes future traffic-analysis attacks much harder to design. Moreover, as HTTPOS provides fundamental defense strategies and basic methods to modify flow features, new evasion methods may be developed based on them. Moreover, we report below our additional findings on web bugs, another attack model for the CWWZ attacks and our solutions to defending against it, and our measurement results for the support rate of HTTP Range.

5.4.1 Web bugs

In the course of conducting the measurement experiments, we observed some cases where the size of packets carrying certain web objects cannot be adjusted. These web objects are usually $1 * 1$ pixel *web bugs* belonging to online advertisement companies that customize their web servers, and none of our methods works for them. Since these web bugs are usually used to track users, a user may just filter them to protect privacy. Moreover, since they may exist in many web pages, their sizes could increase an attack's false positive rate instead of facilitating the attack.

5.4.2 The CWWZ attack

We also note that if an advanced CWWZ attacker can observe the payload of HTTPS packets, she may still be able

to infer the size of a web object even after changing the packet size. More precisely, an attacker can first identify packets carrying SSL/TLS application data from the type field in the SSL/TLS header and then use the field of application data length to assemble consecutive TCP packets. However, such attack does not work if the SSL/TLS packets go through an IPSec tunnel or a wireless channel.

HTTPOS tackles this attack through two approaches. The first approach is for the servers that support HTTP Range. We measured the support rate of HTTP Range by HTTPS servers and found that more than 80% URLs in the measured servers support HTTP Range. Further details are given in Section 5.4.3. In this case, HTTPOS first divides the web object into a random number of portions and makes these portions to overlap a random part with one another. In this way, the web object sizes reported by the field of application data length in the SSL/TLS header are incorrect. The following is an example of downloading a web object from Twitter through HTTPS. In the normal case, the packet size sequence is $(476 \rightarrow, 1460 \leftarrow, 1460 \leftarrow, 1171 \leftarrow)$, and the SSL/TLS record size sequence is $(471 \rightarrow, 4086 \leftarrow)$. Although we can use TCP-based methods to split packets, the attacker may still infer the response size from the sequence of the SSL/TLS records' sizes. After HTTPOS applies HTTP Range, the sequence of packet sizes is modified to $(497 \rightarrow, 1460 \leftarrow, 245 \leftarrow, 500 \rightarrow, 1460 \leftarrow, 258 \leftarrow, 500 \rightarrow, 1460 \leftarrow, 254 \leftarrow)$, and the sequence of the SSL/TLS records' sizes becomes $(492 \rightarrow, 1700 \leftarrow, 495 \rightarrow, 1713 \leftarrow, 495 \rightarrow, 1709 \leftarrow)$. As a result, the attacker is prevented from recovering the response packet size.

The second approach is for the servers that do not support HTTP Range (e.g., the Google HTTPS-based search service). In this case, HTTPOS can still inject a number of useless requests from the ambiguity set and set each request message to the same size. This strategy is motivated by the observation that an attacker could not know the exact length of the word typed by a user, and the inserted requests result in many possible words. For example, if a user types "hi," an attacker can observe the following packet size sequence $(539 \pm 20 \rightarrow, 679 \leftarrow, 540 \pm 20 \rightarrow, 658 \leftarrow)$ and the SSL/TLS record sequence $(534 \pm 20 \rightarrow, 291 \leftarrow, 378 \leftarrow, 535 \pm 20 \rightarrow, 291 \leftarrow, 357 \leftarrow)$, where 291 is the length of the HTTP response header. Once HTTPOS inserts useless requests sequence "card" through HTTP pipelining, the packet size sequence and SSL/TLS record sequence become $(1418 \rightarrow, 165 \rightarrow, 1418 \leftarrow, 578 \leftarrow, 1418 \rightarrow, 165 \rightarrow, 1418 \leftarrow, 530 \leftarrow)$ and $(1578 \rightarrow, 291 \leftarrow, 378 \leftarrow, 291 \leftarrow, 355 \leftarrow, 291 \leftarrow, 361 \leftarrow, 1578 \rightarrow, 291 \leftarrow, 352 \leftarrow, 291 \leftarrow, 357 \leftarrow, 291 \leftarrow, 336 \leftarrow)$. Based on the packet size sequence, the CWWZ attack's reduction power is reduced to one, because the sequence has been totally changed.

Although the reduction power of an advanced CWWZ attack exploiting the SSL/TLS record sequence could not be reduced to one, the attacker still could not know the user's input, because there are at least six possible words, including "h," "hi," "c," "ca," "car," and "card." Note that any word that can result in the same SSL/TLS record sequence as any one of the six words is also a possible candidate. For example, if word "x" and "y" induce the same SSL/TLS record size as word "h," both "x" and "y" will be considered as possible inputs by the attack. Therefore, when more useless requests are injected, it becomes harder for such attacks to recover the original sequence.

Since injecting useless requests may introduce much overhead, another approach for evading the CWWZ attack targeting auto-suggestion is to send only one request with all inputs. For example, if a user inputs "hi" in the search box, the auto-suggestion function may send the first request packet with "h" and then the second packet with "hi." To evade the CWWZ attack, HTTPOS may just transmit the request carrying "hi" but drop the request carrying "h." However, this approach may affect web usability.

5.4.3 Support rate of HTTP Range by HTTPS servers

We measured the support rate of HTTP Range for HTTPS servers from two sets of web sites. The first one contains the web sites of the 50 largest banks in America[2]. By using `Pagestats` to visit these web sites' front pages or login pages (if their front pages do not support HTTPS) through HTTPS, we collected 1585 valid URLs from 104 HTTPS servers, and 1323 URLs support HTTP Range (i.e., the support rate is 83.47%). The second data set is based on web sites ranked by Alexa [1]. We first connected to the 443 port (i.e., the default HTTPS service port) of top 1M web sites and stopped when we obtained 3000 web sites, to which the SSL/TLS connections were successfully established. Since not all web sites that open 443 port provide web service through HTTPS, we found only 1245 sites that offer HTTPS-based web service. By crawling their front pages, we gathered $45,401$ URLs from 2448 HTTPS servers, and 85.09% (i.e., $38,632$) of the URLs support HTTP Range.

6 Related work

Most of the existing proposals on defeating against traffic-analysis attacks on encrypted HTTP traffic require modifications to web servers, browsers and/or web objects. In contrast, our HTTPOS is a browser-side solution that does not need such modifications. Moreover, since none of the existing techniques changes all four basic flow features, they could be defeated by the traffic-analysis attacks detailed in Section 3.1. On the other hand, our techniques can successfully evade all of these attacks.

[2] `http://nyjobsource.com/banks.html`

Sun et al. [35] proposed twelve countermeasures and discussed the related costs. Most of them require the support of the web server and/or some modifications to the web objects. One exception is to use HTTP Range to increase the size of web objects. However, since these methods camouflage just the number and the length of web objects, they may not evade the traffic-analysis attacks based on packet size distribution and packet timing information [7,26]. Moreover, except for padding and pipelining, Sun et al. listed the properties of each method but without implementing and evaluating them. On the other hand, HTTPOS exploits protocol features in both TCP and HTTP to change the four basic HTTP flow features. Moreover, we implemented HTTPOS and carefully evaluated the HTTPOS methods on live HTTP traffic.

Hintz [23] and Danezis [15] suggested a number of approaches to evade traffic-analysis attacks, for instance, adding useless data to the flow and removing some web objects in a web page. However, such approaches may not evade new attacks [13, 26], because they do not change the distributions of packet size and timing among packets. Modifying a browser's setting to force all web objects to be transferred through one connection may evade the attack proposed by Coulls et al. [14] that is based on the number of TCP connections belonging to the same web session and the amount of bytes delivered by individual TCP connections. However, the volume of packets from a web server between two requests may still be exploited to identify web sites, because web browsers usually send HTTP requests one after the other, and web servers usually process these HTTP requests in sequence [24].

Wright et al. proposed traffic morphing to evade the traffic-analysis attacks based on packet size and direction [38]. Their system aims at incurring less additional data to a flow while enabling a flow to evade traffic-analysis attacks. Their system first profiles the packet size distribution for each web site and prepares a transformation matrix that maps a packet size in one flow to a packet size in another flow. Before sending a packet, traffic morphing changes the packet size according to the transformation matrix by either splitting or padding the packet. By doing so, the distribution of packet size observed by an attacker will not be the same as the distribution of original packets. Due to padding, traffic morphing may also change the flow size and packets' timing information. However, traffic morphing requires modifications to both web server and web browser, because it needs to pad or split packets on the server side and remove padded stuff on the client side. Moreover, transforming all packets at the server site in real time will affect the performance of the web service. Unfortunately, the authors did not provide the evaluation results. In contrast, HTTPOS does *not* modify both web server and web browser. In fact, a server may not even know whether a client is using HTTPOS. Moreover, we adopt many approaches to mitigate possible performance degradation caused by HTTPOS.

Although anonymity networks (e.g., Tor [17]) also provide anonymous surfing, there are two major differences between HTTPOS and an anonymity network. First, HTTPOS prevents an attacker from inferring the web site a user is visiting, whereas an anonymity network prevents a web server from knowing who is visiting it. Moreover, as pointed out by Sun et al. [35], although multiple proxies are used in anonymity networks, the first link between a client and the first proxy is still vulnerable to those traffic-analysis attacks. Second, anonymity networks are usually provided by a third party, but HTTPOS is a browser-side solution under a user's control.

Information leaks through HTTP covert channels and the corresponding detection mechanisms have been examined recently. Feamster et al. [19] employed sequences of HTTP requests to transmit covert information. Burnett et al. [11] embedded stealthy information into user-generated content. We proposed WebShare [27] to leak information through the value of prevalent web counters. On the defense side, Borders and Prakash designed WebTap [8] to detect HTTP covert channels that convey information through the content and the timing of HTTP requests, and proposed a framework to quantify such information leaks [9]. Schear et al. devised Glavlit [33] to throttle content-based HTTP covert channels. Zhang et al. invented Sidebuster [40] to automatically detect and quantify possible side-channel leaks in web applications.

7 Conclusions

In this paper, we proposed a suite of new browser-side techniques to prevent an attacker from inferring web sites or web pages visited by a user. These techniques exploit the basic protocol features in TCP and HTTP to manipulate four fundamental characteristics in encrypted HTTP flows. We implemented these techniques into a browser-side system called HTTPOS that does not need to modify or control any web entity. An extensive evaluation of HTTPOS on live HTTP traffic shows that it can evade the state-of-the-art attacks with low overhead. Future works include further mitigating the impact of HTTPOS on the performance and sealing other privacy leakages in web browsers.

Acknowledgments

We thank the anonymous reviewers for their quality reviews and David Evans, in particular, for shepherding our paper, and Paul Royal for his suggestions and help. This work is partially supported by a grant (H-ZL17) from The Hong Kong Polytechnic University. This material is based

upon work supported in part by the National Science Foundation under grant no. 0831300, the Department of Homeland Security under contract no. FA8750-08-2-0141, the Office of Naval Research under grants no. N000140710907 and no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

References

[1] Alexa Internet, Inc. `http://www.alexa.com`.

[2] Freebsd/i386 5.2.1-release release notes. `http://www.freebsd.org/releases/5.2.1R/relnotes-i386.html`.

[3] Google trends. `http://www.google.com/trends`.

[4] Google's server names. `http://googlesystem.blogspot.com/2007/09/googles-server-names.html`.

[5] NetCraft Ltd. `http://www.netcraft.com`.

[6] D. Barrett, R. Silverman, and R. Byrnes. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, 2005.

[7] G. Bissias, M. Liberatore, D. Jensen, and B. Levine. Privacy vulnerabilities in encrypted HTTP streams. In *Proc. Privacy Enhancing Technologies Workshop*, 2005.

[8] K. Borders and A. Prakash. Web Tap: Detecting covert web traffic. In *Proc. ACM CCS*, 2004.

[9] K. Borders and A. Prakash. Quantifying information leaks in outbound web traffic. In *Proc. IEEE Symp. Security and Privacy*, 2009.

[10] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho. Seven years and one day: Sketching the evolution of Internet traffic. In *Proc. IEEE INFOCOM*, 2009.

[11] S. Burnett, N. Feamster, and S. Vempala. Chipping away at censorship with user-generated content. In *Proc. USENIX Security*, 2010.

[12] CACE Technologies, Inc. AirPcap. `http://www.cacetech.com`.

[13] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *Proc. IEEE Symp. Security and Privacy*, 2010.

[14] S. Coulls, C. Wright, F. Monrose, M. Collins, and M. Reiter. On web browsing privacy in anonymized NetFlows. In *Proc. USENIX Security*, 2007.

[15] G. Danezis. Traffic analysis of the HTTP protocol over TLS. `http://research.microsoft.com/en-us/um/people/gdane/papers/TLSanon.pdf`, 2007.

[16] S. Dedeo. Pagestats. `http://web.cs.wpi.edu/~cew/pagestats/`.

[17] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. USENIX Security*, 2004.

[18] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[19] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing censorship and surveillance. In *Proc. USENIX Security*, 2002.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.

[21] M. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly Media, 2005.

[22] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive-Bayes classifier. In *Proc. ACM Workshop on Cloud Computing Security*, 2009.

[23] A. Hintz. Fingerprinting websites using traffic analysis. In *Proc. Privacy Enhancing Technologies Workshop*, 2002.

[24] D. Koukis, S. Antonatos, and K. Anagnostakis. On the privacy risks of publishing anonymized IP network traces. In *Proc. IFIP Communications and Multimedia Security*, 2006.

[25] Y. Lee. SOCKS: A protocol for TCP proxy across firewalls. `http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol`.

[26] M. Liberatore and B. Levine. Inferring the source of encrypted HTTP connections. In *Proc. ACM CCS*, 2006.

[27] X. Luo, E. Chan, and R. Chang. Crafting web counters into covert channels. In *Proc. IFIP SEC*, 2007.

[28] X. Luo, E. Chan, and R. Chang. CLACK: A network covert channel based on partial acknowledgment encoding. In *Proc. IEEE ICC*, 2009.

[29] G. Macia, Y. Wang, R. Rodriguez, and A. Kuzmanovic. ISP-enabled behavioral ad targeting without deep packet inspection. In *Proc. IEEE INFOCOM*, 2010.

[30] NetCraft Ltd. November 2010 web server survey. `http://news.netcraft.com/archives/2010/11/05/november-2010-web-server-survey.html`.

[31] Northrop Grumman Corp. Cloc. `http://cloc.sourceforge.net/`.

[32] M. Ruef. httprecon. `http://www.computec.ch/projekte/httprecon/`.

[33] N. Schear, C. Kintana, Q. Zhang, and A. Vahdat. Glavlit: Preventing exfiltration at wire speed. In *Proc. HotNets-V*, 2006.

[34] R. Sinha, C. Papadopoulos, and J. Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC/Information Sciences Institute, 2007.

[35] Q. Sun, D. Simon, Y. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proc. IEEE Symp. Security and Privacy*, 2002.

[36] The University of Waikato. Weka. `http://www.cs.waikato.ac.nz/~ml/weka/`.

[37] P. Wouters and K. Bantoft. *Openswan: Building and Integrating Virtual Private Networks*. Packt Publishing, 2006.

[38] C. Wright, S. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proc. ISOC NDSS*, 2009.

[39] T. Yen, X. Huang, F. Monrose, and M. Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In *Proc. DIMVA*, 2009.

[40] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *Proc. ACM CCS*, 2010.