EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Efficient, Effective, and Realistic Website Fingerprinting Mitigation

Weiqi Cui[1], Jiangmin Yu[1], Yanmin Gong[2], Eric Chan-Tin[3,*]

[1]Oklahoma State University
[2]University of Texas - San Antonio
[3]Loyola University Chicago

Abstract

Website fingerprinting attacks have been shown to be able to predict the website visited even if the network connection is encrypted and anonymized. These attacks have achieved accuracies as high as 92%. Mitigations to these attacks are using cover/decoy network traffic to add noise, padding to ensure all the network packets are the same size, and introducing network delays to confuse an adversary. Although these mitigations have been shown to be effective, reducing the accuracy to 10%, the overhead is high. The latency overhead is above 100% and the bandwidth overhead is at least 30%. We introduce a new realistic cover traffic algorithm, based on a user's previous network traffic, to mitigate website fingerprinting attacks. In simulations, our algorithm reduces the accuracy of attacks to 14% with zero latency overhead and about 20% bandwidth overhead. In real-world experiments, our algorithms reduces the accuracy of attacks to 16% with only 20% bandwidth overhead.

Received on 30 February 2019; accepted on 20 April 2019; published on 29 April 2019

Keywords: Privacy, Noise, Website Fingerprinting, Cover Traffic

doi:10.4108/eai.29-1-2019.161977

1. Introduction

Website fingerprinting violates the privacy expected from a user when she is using an anonymizing service such as a proxy or Tor [1]. The goal of website fingerprinting attacks [2] is to determine the website visited by a victim. The adversary, in this case, is usually local, for example on the same network or the Internet Service Provider, and can observe all the network traffic sent by the victim. These attacks are effective and are accurate in successfully identifying the websites. The accuracy is over 90% even when the network traffic is encrypted or anonymized through a proxy. Since the adversary knows who the user is and can accurately guess what websites she is visiting, the user has no privacy.

Various defenses against website fingerprinting attacks [3–7] have been proposed. The defenses include padding so that every packet has the same size, cover traffic to generate enough noise to fool the adversary, or introducing network delays between network packets. Although they have been shown to be effective, the overhead introduced by these defenses is high. The latency overhead is above 100% and the bandwidth overhead is from 30% to over 100%.

Our **contribution** is a new cover traffic algorithm that generates just enough noise to mitigate website fingerprinting attacks. Our algorithm also has zero latency overhead and lower bandwidth overhead than current schemes. Our algorithm generates "realistic" cover traffic [1]; it collects the network traffic from a user, then uses that historical network traffic data as training set to feed the cover traffic generation algorithm. The generated noise thus will look exactly like a website that a user has previously visited. This prevents website fingerprinting attacks and introduces little bandwidth overhead. In a workshop paper [8], we showed through simulations that our proposed algorithm reduces accuracy of attacks to 14% while

[*] Corresponding author. Email: Chantin@cs.luc.edu

[1] cover traffic and noise are used interchangeably

introducing no latency overhead and 20% bandwidth overhead. We expand this work by undertaking real world experiments and show that our simulation results hold in these experiments.

Table 4 shows a comparison of our proposed algorithm with existing mitigation techniques. Our algorithm has comparable accuracy over the other schemes, zero latency overhead, and lower bandwidth overhead. The table shows the lowest accuracy (best-case for the mitigation) regardless of the classification algorithm used. Our algorithm has zero latency overhead since we are only introducing cover traffic. No delays are introduced.

The remainder of this paper is organized as follows. Section 2 gives a survey of related work. Section 3 describes website fingerprinting attacks and our threat model. The design of our proposed cover traffic algorithm is provided in Section 4. The simulated experiments and results are outlined in Section 5. Section 6 shows the effectiveness of our proposed algorithm in the real-world. Future work is discussed in Section 7.

2. Related Work

It has been shown that analyzing encrypted network traffic can reveal the websites and webpages visited [2–6, 9–21]. Since the payload is encrypted, only the metadata is available such as packet sizes, number of packets, direction of packets, and time interval between packets. A training dataset is built. Then, given a network traffic trace, machine learning techniques are used to predict the website visited. Previous results have shown that websites can be recognized with a high accuracy. More recent research results have looked at anonymized network traces such as using Tor instead of a simple HTTPS proxy. Although initial results showed that Tor provided adequate protection against website fingerprinting, more advanced data parsing techniques show that websites can be recognized with a fairly high accuracy even when the website trace is over Tor. The consequences of website fingerprinting is censorship or prosecution by the government if the user visits a forbidden website. It has been argued [22] that website fingerprinting is not a practical attack due to the large number of webpages and the false positive would be high. Website fingerprinting attacks have also been extended to identify the webbrowser used [23], which could lead to user identification as most users utilize a unique webbrowser [24].

Website fingerprinting is one type of network traffic analysis. There has been other work on network traffic analysis [25] and traffic analysis resistant protocols [26–28]. Network traffic analysis is usually performed for censorship [29]. Various techniques to avoid censorship have been proposed, using traffic morphing [30] to disguise the network traffic as VoIP [31, 32] or using other covert channels [33–35]. It has, however, been shown that it is still possible to see through this obfuscation [36–38].

Using cover/dummy/fake traffic to mask a user's activities has been proposed before [39]. It has been shown that this mechanism can be countered or the cover traffic removed [40, 41] to reveal the user's activities. Cover traffic is useful to mask real web search queries by performing many other unrelated and random search queries. Cover traffic can also be used to make network traffic analysis harder by adding unrelated network-level packets. Our algorithm generates realistic cover traffic making it harder for website fingerprinting attacks to accurately guess the website from the observed packet trace. Another scheme, Track Me Not [42], focused on web search queries and generating fake web searches, but [43] has shown that web search queries obfuscation can still be analyzed.

Various website fingerprinting defenses have been proposed [3–7, 14, 44, 45]. They all make use of some sort of padding, delaying sending of packets, or adding cover traffic. Many of these defenses have high latency and/or bandwidth overhead and have been shown to be effective in mitigating website fingerprinting attacks. Our proposed defense has zero latency overhead and lower bandwidth overhead while maintaining a high level of effectiveness.

Traffic morphing [46] is another possible defense against website fingerprinting attacks. It attempts to modify the shape and patterns of network traffic such that it looks different. For example, Stegotorus [47] attemps to make Tor network traffic look like HTTPS. Similarly, [31, 32] attempt to morph Tor traffic to look like VoIP traffic so that network traffic analysis or deep packet inspection will not allow Tor traffic to be blocked or identified; VoIP traffic is usually allowed. However, [37, 38] have shown that these traffic morphing schemes can be circumvented. We are not proposing to modify network traffic patterns. Our algorithm generates realistic cover traffic to mask the original packet trace.

3. Background

3.1. Website Fingerprinting

Website fingerprinting aims to determine the website visited by examining the network trace sent by a victim's webbrowser. That trace is usually encrypted and sent over a proxy or an anonymous network like Tor [1, 48] so that the network contents cannot be analyzed. The only information that can be observed are the packet sizes, the direction of the packets, the time interval between packets, and the number of packets sent and received.

A packet trace PT consists of n network packets. PT also consists of the tuple $< \Sigma_{i=0}^{n} N_i, N_s, N_c >$, where N_i is each individual network packet, N_s is the total number of packets from server to client, N_c is the total number of packets sent from the client to the server. Each network packet N_i forms the tuple $< S_i, T_i, D, M_s, M_c >$, where S_i is the size of the packet, T_i is the time interval until the next packet, D is the direction of the network packet (from client to server or from server to client), M_s is the number of packets from the server to the client, and M_c is the number of packets from the client to the server. M_s and M_c denotes each "train" of packets, that is, there are usually a few packets from client to the server, followed by several packets from server to client. On the other hand, N_c and N_s denotes the total number of packets for the whole packet trace.

3.2. Classification

Previous work [2–4, 9–16, 18–21] have achieved a classification accuracy of around 90% in both the open and closed world settings. A closed world is where the set of training packet traces are the same as the testing set. An open world setting is where there is a small set of monitored/sensitive packet traces among a larger set; the goal is to detect if a packet trace belongs to one of these monitored websites.

To perform classification, various features have been used such as number of outgoing and incoming packets, total size of incoming and outgoing packets, and cumulative size of packets. If the Tor network is used, some features that have also been considered include the Tor cells before and after. Various algorithms have also been used such as k-nearest neighbors (K-NN), support vector machine (SVM), random decision forests, edit distances, Jacard index, and Naive Bayes.

In this work, we used the same features as those mentioned in [18]: cumulative size of packets sampled at regular intervals over the whole packet trace, number of incoming packets, total size of incoming packets, number of outgoing packets, and total size of outgoing packets. We also used the random decision forests as it was used by the most recent paper [21].

3.3. Mitigations

Various defenses against website fingerprinting attacks have been proposed such as padding of packets to a fixed size and cover traffic (noise) to mask the real packet trace. They have all been shown to be somewhat effective reducing the accuracy of website fingerprinting to about 10% to 30%. However, all of these defenses incur high latency overhead or high bandwidth overhead or both.

Our proposed algorithm achieves a similar reduction in accuracy while keeping the overhead manageable. Our cover traffic generated is realistic instead of random as it depends on what the user has done previously. A threshold value for the amount of cover traffic generated can also be chosen by the user. Our algorithm experiences zero latency overhead.

3.4. Threat Model

The threat model is a local adversary that can see all the network traffic from a user. The adversary cannot decrypt the contents of the network packets but can observe the metadata such as packet sizes, direction of the packets, and the timings of packets. The adversary can also look at the IP headers to determine the source and destination IP addresses and port numbers. The victim is using an anonymous service such as a VPN or Tor [1]. Figure 1 illustrates the model and shows where the adversary is located. The goal for the adversary is to guess the website or webpage from only the encrypted network packet trace.

4. Proposed Noise Algorithm

4.1. Overview

Our proposed algorithm to generate cover traffic is novel since it generates realistic noise rather than random noise or random padding. The intuition of this algorithm is from the results of previous work [19] which shows that it's hard to split two mixed traces of websites. The noise generated is learned from the network traffic generated by the user's webbrowser. The information recorded is the network traffic trace without the payload contents: each incoming and outgoing packet's size, and the time interval between packets and "train" of packets. A train is a set of incoming packets with size of MTU (Maximum Transmission Unit) with the last packet size less than MTU. Usually an outgoing web request is followed by one or more trains of incoming packets. Replaying this recorded network traffic will simulate that user's browsing habit. Our hypothesis is that if the cover traffic generated is similar to what the user usually does, this will provide a better noise in preventing website fingerprinting and also reduce the bandwidth overhead since this would be traffic that the user usually generates anyway. It has already been shown that if a client visits several webpages at the same time [19], then it is hard for an adversary to identify the webpage visited.

Instead of replaying the web requests to the actual servers which would use up resources on these servers, we set up our own simple webserver. Our algorithm can be implemented as a plugin for Firefox (Tor Browser Bundle). It will send a web request padded to a certain packet size to our webserver through the Tor network. The request will contain the total size of data that the server has to send back and the time that the data

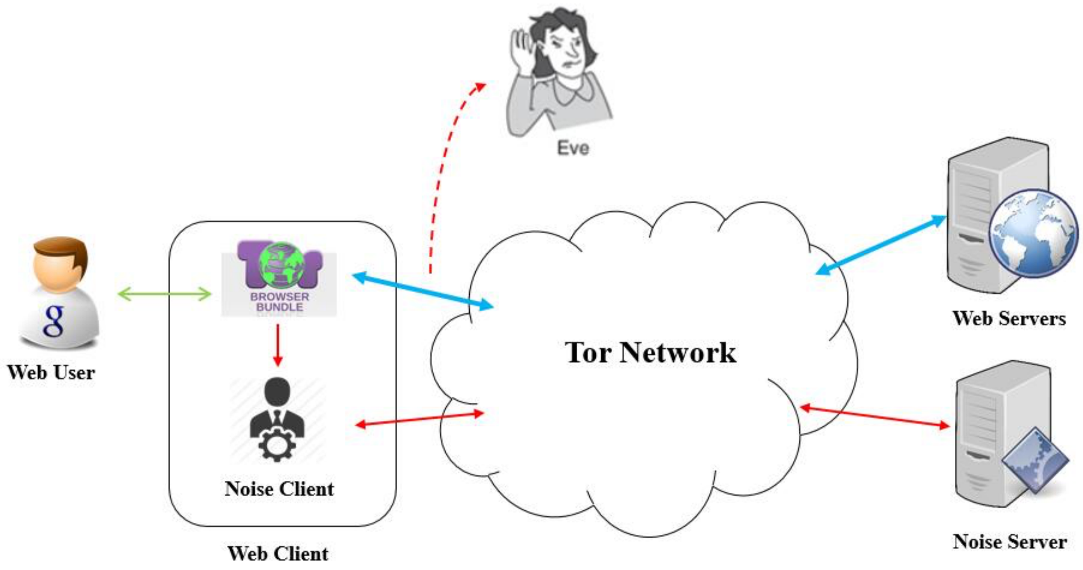


Figure 1. Our system model and experimental setup.

should be sent. Both the client plugin and webserver do not have to send any content; only pad the packets to the specified packet size. The generated network traffic will be transferred over the Tor network; a local adversary will not be able to determine which packet is noise.

The algorithm will first record traffic of a web page, then parses the recorded traffic trace. Packets are put into two sets based on whether they are incoming packets or outgoing packets. For each set, packets are organized by trains of packets. For each train, the size and timestamp of each packet is recorded. Trains of packets are listed in order by the timestamp of the first packet in the train.

Figure 2 shows an example sample of a recorded network traffic. The only information recorded is the relative time between packets and the packet sizes. Both incoming and outgoing packets are recorded. The format of the sample shown in the figure is as follows: $< timestamp >:< packetsize >$. The actual time is not relevant; only the time difference between two packets is used. This is the time difference between the current packet's timestamp and the next packet's timestamp. The packet size is the TCP-level packet size. A red packet size indicates an outgoing packet.

4.2. Pseudocode for Algorithm

Ideally, the generated cover traffic will be exactly the same as the real recorded sample. For example, they have the same number of packets and the same timing gap between packets. In practice, we cannot control the time distribution of every packet due to network latency. Instead, we break the sample traffic into several segments based on the timing gap of packets. Then our goal is to make the cover traffic have similar timing gaps with the sample in terms of traffic segments. The pseudocode of noise client and server is shown in Algorithm 4.2 and Algorithm 4.2.

FnFunction isend FRecursconstructNoiseRequestList *[h]Construct noise request list $\{Req_{i1}, Req_{i2} ...... Req_{in}\}$ based on S_i

*[h]S_i is the size of each packet.

()S_i

*[h]Break S_i into segments based on MaxNumberOfRequest and MinTimediffBtwRequests parameter

$\{RS_{i1}, RS_{i2} ...... RS_{in}\} \leftarrow break(S_i, MaxNumberOfRequest, MinTimediffBtwRequests)$

*[h]RS_{ij} is the size of the packet for each request to the noise server.

RS_{ij} in $\{RS_{i1}, RS_{i2} ...... RS_{in}\}$ $sumOfOutgoingPktSize \leftarrow sum(outgoing\ packet's\ size)$ $timeOfRequest \leftarrow time\ of\ first\ outgoing\ packet$ $Req_{ij} \leftarrow new\ Req$(timeOfRequest, sumOfOutgoingPktSize)

*[h]For each request, generate the number of response packets PRS_{ijk} and the size of each packet

$\{PRS_{ij1}, PRS_{ij2}, ..., PRS_{ijn}\} \leftarrow$ break(RS_{ij}, $MaxNumberOfResponse$, $MinTimediffBtwResponse$)

PRS_{ijk} in $\{PRS_{ij1}, PRS_{ij2}, ..., PRS_{ijn}\}$ $sumOfSubResponsePktSize \leftarrow$

$sum(incoming \quad packets \quad size) * CoverTrafficLoadRatio$ $timeOfSubResponse \leftarrow time \quad of \quad the \quad first \quad incoming \quad packet$ $Req_{ij}.append(timeOfSubResponse : sumOfSubResponsePktSize)$ $\{Req_{i1}, Req_{i2}, ..., Req_{in}\}.append(Req_{ij})$ $\{Req_{i1}$, Req_{i2} ...... $Req_{in}\}$ sNoiseRsendNoiseRequest ()$ReqList$ Req_{ij} in $\{Req_{i1}$, Req_{i2} , . . . , $Req_{in}\}$ *[h]Send Req_{ij} to noise server. Wait and receive message from noise server fork(Send Req_{ij} to noise server. Wait and receive message from noise server.)

*[h]Sleep certain time based on the time gap between Req_{ij} and $Req_{i(j+1)}$ $sleep()$ PrProcedure isend Pr $mainRecordsampletraffictracesfornwebpages, denotedas\{S_1,$ S_2 ...... $S_n\}$ $S_i \leftarrow random(\{S_1,$ S_2 ...... $S_n\})$ $ReqList \leftarrow constructNoiseRequest(S_i)$ $sendNoiseRequest(Reqlist)$

FnFunction isend FRecursforkNewThread (*[h])Req_{ij}

*[h]For each request RS_{ij} from the client, respond with number and size of response packets based on PRS_{ijk}

$\{RS_{ij1}$, RS_{ij2} ...... $RS_{ijn}\}\leftarrow extract(RS_{ij})$

PRS_{ijk} in $\{PRS_{ij1}$, PRS_{ij2} , . . . , $PRS_{ijn}\}$ *[h]Send random characters with specific size to client

$send(randomcharacters, client)$

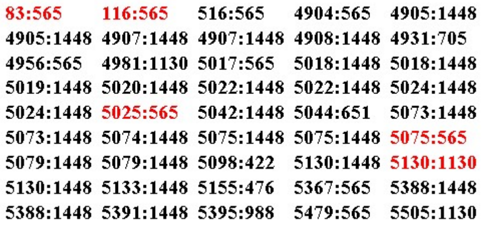
*[h]sleep certain time based on the time gap between $PReq_{ijk}$ and $PReq_{ij(k+1)}$

$sleep()$ True $Req_{ij} \leftarrow receive()$

$forkNewThread(Req_{ij})$

4.3. Implementation Details

The cover traffic needs to have two features: 1) it should customize total packet size so that the cover traffic can be controlled, 2) it should have similar packet distribution with the real web traffic so that it cannot be filtered out easily. Modern web pages usually contain multiple resources, such as html text, CSS files, javascript files, and images. Modern web browsers support multiple parallel connections to a website. From a typical web page network traffic, we can see that a web browser sends multiple requests to download multiple resources in parallel. Considering these web page features, we separate web page traffic into segments based on requests. We define two parameters: 1) *MaxNumberOfRequest* and 2) *MinTimediffBtwRequests*. *MaxNumberOfRequest* denotes the maximum number of web requests and *MinTimediffBtwRequests* denotes the minimum time interval between two web requests. Combining these two parameters, we can separate a network traffic trace into request segments. We scan the recorded traffic trace, find two consecutive outgoing packets which have a time interval greater than *MinTimediffBtwRequests*, then use these two packets as the delimiter of two request segments. If we get a higher number of request segments than *MaxNumberOfRequest*, the algorithm will adjust the requested segment time interval threshold accordingly to get exactly *MaxNumberOfRequest* number of request segments. To further control the distribution of incoming packets, we separate incoming packets into segments inside the request segment. We define two other parameters: 1) *MaxNumberOfResponse* and *MinTimediffBtwResponses*. These two parameters work the same as for outgoing packets.



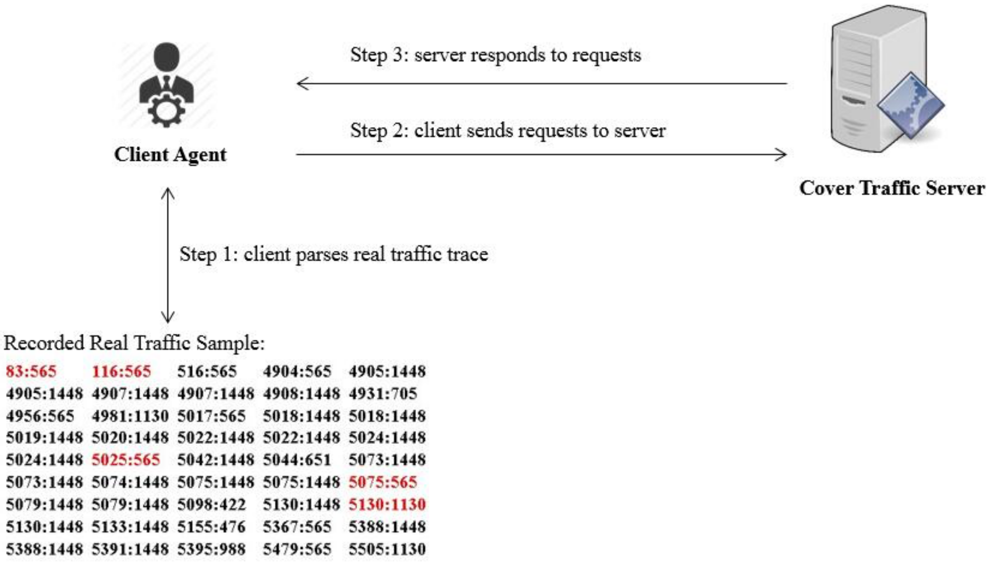
```
83:565     116:565    516:565    4904:565   4905:1448
4905:1448  4907:1448  4907:1448  4908:1448  4931:705
4956:565   4981:1130  5017:565   5018:1448  5018:1448
5019:1448  5020:1448  5022:1448  5022:1448  5024:1448
5024:1448  5025:565   5042:1448  5044:651   5073:1448
5073:1448  5074:1448  5075:1448  5075:1448  5075:565
5079:1448  5079:1448  5098:422   5130:1448  5130:1130
5130:1448  5133:1448  5155:476   5367:565   5388:1448
5388:1448  5391:1448  5395:988   5479:565   5505:1130
```

Figure 2. Sample of recorded network traffic. The format is $< timestamp >:< packetsize >$. Red indicates outgoing packets.

To control the **overall** cover traffic size, we define another parameter *CoverTrafficLoadRatio*. The total cover traffic size will be the total packet size of network traffic multiplied by *CoverTrafficLoadRatio*. Since we separate the network traffic trace into segments, the size of cover traffic segments will be the size of corresponding traffic segments multiplied by *CoverTrafficLoadRatio*.

So far, we have parsed the recorded real web traffic, our next step is to generate the cover traffic. To do that, we have a cover traffic client agent running on the user side. This client agent connects to a cover traffic server, sends requests to the server and receives responses from the server. Figure 3 shows how our cover traffic is generated. The first step we need to do is to parse the recorded real traffic trace. As we can see from the sample traffic trace, the only information recorded is the time of packets and the packet sizes. Both incoming and outgoing packets are recorded. The format of the sample shown in the figure is as follows: $< timestamp >:< packetsize >$. We use negative packet sizes for outgoing packets and positive packet sizes for incoming packets.

The content of a request contains the following information: relative time to send back segments of responses and total packet size of each segment. Table 1 shows a list of requests sent to the cover traffic server. These requests are generated by parsing one recorded web traffic network trace. Each cover traffic request will be sent at a certain time ("Time to Send") and will be of certain size ("Total Size"). The request content shows the number of responses to be sent back by the server,


Step 3: server responds to requests
Step 2: client sends requests to server
Client Agent
Cover Traffic Server
Step 1: client parses real traffic trace

Recorded Real Traffic Sample:

83:565	116:565	516:565	4904:565	4905:1448
4905:1448	4907:1448	4907:1448	4908:1448	4931:705
4956:565	4981:1130	5017:565	5018:1448	5018:1448
5019:1448	5020:1448	5022:1448	5022:1448	5024:1448
5024:1448	5025:565	5042:1448	5044:651	5073:1448
5073:1448	5074:1448	5075:1448	5075:1448	5075:565
5079:1448	5079:1448	5098:422	5130:1448	5130:1130
5130:1448	5133:1448	5155:476	5367:565	5388:1448
5388:1448	5391:1448	5395:988	5479:565	5505:1130

Figure 3. Cover traffic client and server.

along with the time and packet size of each response. Since all requests and responses are encrypted, no actual content is sent; the content of both the request and response can be filled with random data.

From Table 1, the client agent sends a total of nine requests to the cover traffic server. The first request is sent at time 0 (relative time), the size of the request is 703 bytes and the request content is "Response=0:676,354:18,756:91". We add padding to the request content if its size is less then the expected request size. When the cover traffic server receives this request, it will send back 676 bytes of data at time 0. The time is relative to when the server receives the request. At time 0, that means the server just received the request. At time 354 ms, the server sends a response packet of size 18 bytes and at time 756 ms, the server sends a packet response of size 91 bytes. The contents of the response packet the server sends back to the client are filled with random characters. At time 7, 320 ms, the client agent sends the second request to the server.

4.4. Example

Taking the packet traces from Figure 2 as an example, the following two tables are built based on this example. Table 2 shows the parsed outgoing packets set, denoted as TS_{out}. Table 3 shows the parsed incoming packets set, denoted as TS_{in}. Most webbrowsing network traces have a higher number of incoming packets than outgoing packets. Moreover, the size of the incoming packets is higher than outgoing packets, which are usually web requests for a URL resource (such as jpg, html, etc...). This is typical of web traffic and is reflected in the tables. Generating noisy cover traffic works as follows.

1. Randomly select one traffic train from TS_{out} and TS_{in} each, denoted as T_{out} and T_{in} respectively. The total size of T_{out} is S_{out} and the total size of T_{in} is S_{in}.

2. Construct a cover traffic request with size S_{out}.

3. Send this request to the noise server.

4. The server will reply back with data of size S_{in} in WT_{in} milliseconds. WT_{in} is the time difference between the first packet of T_{in} and the first packet of the next traffic train after T_{in} in the incoming traffic train set.

5. Wait for some time WT_{out}, where WT_{out} is the time difference between the first packet of the chosen outgoing traffic train T_{out} and the first packet of the next outgoing traffic train.

6. Repeat steps 1 to 5 until the total incoming traffic from the noise server is equal to the size of all the incoming packets of the recorded traffic trace.

As an example, let's suppose outgoing traffic train 4 is selected from TS_{out} and incoming traffic train 8 is selected from TS_{in}. Our algorithm will create a new cover traffic request to send to the noise server. The request will ask the server to send back data of size $1448 + 1448 + 1448 + 476 = 4820$ bytes with a time of $5367 - 5130 = 237$ milliseconds. The request contains only total size of data to be sent and the time. For

Cover Traffic Request	Time to Send (ms)	Total Size (bytes)	Request Content
1	0	703	Response=0:676,354:18,756:91;
2	7,320	4,149	Response=0:784,872:325;
3	9,190	8,087	Response=0:325,569:1045,1065:645,2000:325, 6109:211;
4	19,196	1,0628	Response=0:217,380:108,657:942,1101:4581, 1444:942,1920:8442;
5	27,094	703	Response=0:676,312:109;
6	29,434	148	Response=0:676;
7	30,395	358	Response=0:18;
8	31,166	197	Response=0:91;
9	38,185	543	Response=0:108;

Table 1. Cover traffic requests based on one recorded real web traffic trace. The request content contains the number of responses to be sent from the cover traffic server and is in the format $< relativetime >$:$< packetsize >$.

Traffic Train ID	Time and Packets
1	83:565
2	116:565
3	5025:565
4	5075:565
5	5130:1130

Table 2. The parsed outgoing traffic train set.

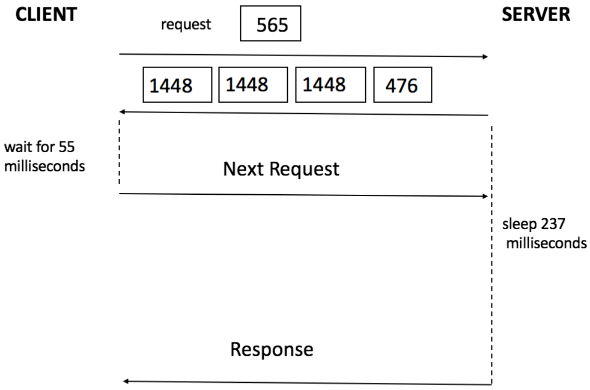


Figure 4. An example of the algorithm, train 4 is selected from TS_{out} and train 8 is selected from TS_{in}.

example, the server only needs to send padded data with size 4820. The lower level network interface will determine how to send each packet – if the MTU is 1448, packet size will be 1448 + 1448 + 1448 + 476. The outgoing packet will be of size 565 bytes. Since the actual contents of the packet is small, the rest of the packet is padded. To simplify the example, we ignore packet headers. When the noise server receives this cover traffic request, it will send back data of size 4820 bytes and sleep for 237 milliseconds before responding to next request. At the same time, the client side waits for 55 milliseconds, which is the time difference between the first packets of the outgoing traffic train 4 and outgoing traffic train 5 from Table 2. The procedure of this example is shown in figure 4. This process is repeated until the sum of all the incoming packet sizes from the server is equal to the recorded traffic trace. The reason for waiting on both client and server sides is to ensure that the generated noise traffic is well distributed to look more realistic. This generated cover traffic can achieve better performance in terms of obfuscating the overall traffic collected by a website fingerprinting attacker.

The user can choose as a parameter, the size of the cover traffic, *CoverTrafficLoadRatio*, s. Since the cover traffic mimics the websites that the user has previously visited, the bandwidth used will be doubled. To minimize the bandwidth overhead, each train size could be reduced by a factor of s. If the factor s is 0.5, the incoming train will thus have a total packet size of $0.5 \times S_{in}$ in time WT_{in}. This reduces the bandwidth overhead and generates fewer packets.

We emphasize that the cover traffic is only between the client's webbrowser and the cover traffic webserver through Tor. The only data sent are padded data so that the packets are of a pre-determined size. The cover traffic generated will look realistic as it is traffic that was generated by the user. This recorded network traffic is only stored locally on the browser.

We expect our algorithm to effectively mitigate website fingerprinting attack since it has already been shown that cover traffic is effective. We expect that our algorithm will have lower bandwidth overhead since the amount of noise generated can be modified. Moreover, there is no extra latency added as no network delay is introduced. Our algorithm only generates cover traffic to another website.

Traffic Train ID	Time and Packets
1	516:565
2	4904:565 4905:1448 4905:1448 4907:1448 4907:1448 4908:1448 4931:705
3	4956:565
4	4981:1130
5	5017:565 5018:1448 5018:1448 5019:1448 5020:1448 5022:1448 5022:1448 5024:1448 5024:1448
6	5042:1448 5044:651
7	5073:1448 5073:1448 5074:1448 5075:1448 5075:1448 5079:1448 5079:1448 5098:422
8	5130:1448 5130:1448 5133:1448 5155:476
9	5367:565
10	5388:1448 5388:1448 5391:1448 5395:988
11	5479:565
12	5505:1130

Table 3. The parsed incoming traffic train set.

5. Simulations

5.1. Setup

We utilized the dataset from [18], which consists of 1,125 webpages and 40 instances of each webpage. Each instance contains the timestamp of each packet with the packet sizes (negative packet sizes indicate outgoing packet). We implemented the noise generation algorithm described in Section 4.

We use the standard Weka [49] tool and experimented with the Random Forest classification algorithm. The classification features used in our experiments are the same as those used in [18]. The first four features are: total size of outgoing packets, total size of incoming packets, total number of outgoing packets, and total number of incoming packets. The remaining features are the sampled cumulative representation of packet size. There are two ways to calculate the cumulative packet size: c is the cumulative size of packets size where an outgoing packet has a negative packet size and a is the cumulative size of packets size where both outgoing and incoming packet sizes are denoted as positive numbers. The number of samples used n can be varied and will be taken at equidistant points in the packet trace. For example, if there are 75 packets and $n = 100$, a sample is taken every 0.75 packet. To determine the packet size of the 0.75th packet, the linear interpolation is calculated. If the 0th packet size was 10 and the 1st packet size was 20, the cumulative packet size for 0 is 10 and the cumulative packet size for 1 is $20 + 10 = 30$. The 0.75th packet size is thus $(0.75 * (30 - 10)) + 10 = 25$.

We compared our proposed cover traffic algorithm with the basic cover traffic scheme. The latter works as follows. When a user visits a website, the basic scheme will randomly pick another website to also visit. As shown by [19], having two simultaneous website visits significantly lowers website fingerprinting accuracy.

The original dataset contained 1,125 webpages, many from the same website. We filtered out webpages of the same website and used 91 websites as our base training dataset [2]. The dataset [18] contained timestamps and packet sizes. Merging the original website packet trace with the noise packet trace is relatively straightforward. Since there are 40 instances of each website, we randomly picked one instance as the noise data to merge with the original packet trace.

We considered two different basic cover traffic algorithms as a comparison. The first one always picks the same webpage (but possibly different instances). The second one randomly picks from a set of 10 webpages, different from the 91 previously selected. The second case provides a more diverse set of webpages and noise to be added.

Our noise generation algorithm "learning" dataset consists of a further 10 webpages where the packet traces are recorded. As the noise is learned from the users' network traffic, in the simulation, we ran our algorithm to generate one packet trace of noise for each of the original 91 webpages and merge that trace with the original webpage packet trace.

5.2. Results

The accuracy of the Random Forest classification algorithm including either features c or features a and c is 81.77% and 81.88% respectively. The first four features are always included. Adding features a, the cumulative total of the absolute value of all packet sizes,

[2] Note that since each webpage is a unique website, we used webpage and website interchangeably from now on.

does not improve the accuracy of website fingerprinting attacks by much. The sample size n was set to the recommended 100 from [18]. We only considered set of features c from now.

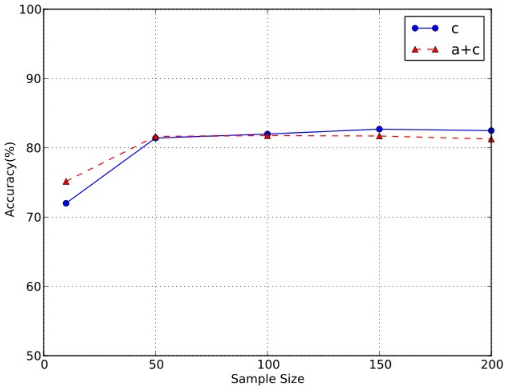


Figure 5. The accuracy using the Random Forest algorithm when varying the sample size. Note the y-axis does not start at 0.

Figure 6 shows the classification accuracy for varying amount of noise added to original traces. Figure 7 shows the bandwidth overhead in % of the extra network traffic generated. The two basic cover traffic algorithms are indicated by $k = 1$ for adding the same one website as noise each time and by $k = 10$ for randomly adding one of ten websites as noise. The x-axis indicates the amount of noise s added. When $s = 1.0$, this means the whole packet trace is added as noise. When $s = 0.5$, only half of the packet trace is added as noise, that is, every other packet is added as noise to preserve the time intervals. For the basic cover traffic cases ($k = 1$ and $k = 10$), we are "simulating" the noise generated; in a real-world setting, this would be hard to achieve without controlling the server – in this case, the browser could send random packets. We show different values of s to compare with our algorithm. As more noise is added (s increases), the accuracy decreases, as expected. Similarly, the bandwidth overhead also increases as more noise is added. Our proposed noise generation algorithm achieves the same accuracy regardless of the amount of noise; this is because we are generating realistic noise that can more effectively hide a user's real traffic rather than generating random noise. Our algorithm's bandwidth overhead is the same as the basic cases. However, even with $s = 0.25$, the overhead is 20% and the accuracy is 14%. Since our proposed algorithm generates random packet traces based on real recorded network traffic, we ran our experiments five times; the graphs show the average of the five experiments. For these experiments, the training dataset used in the Random Forest classification algorithm is the original 91 webpages and the testing dataset is the new webpages with noise added.

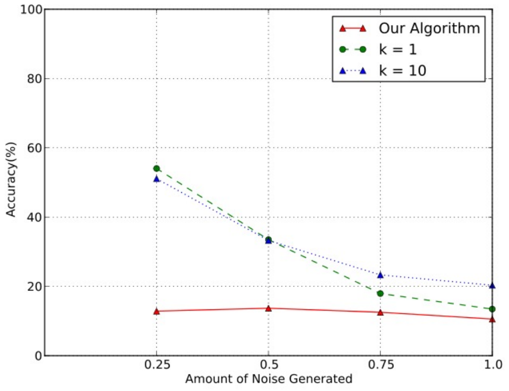


Figure 6. The accuracy using the Random Forest algorithm when introducing different kinds of cover traffic.

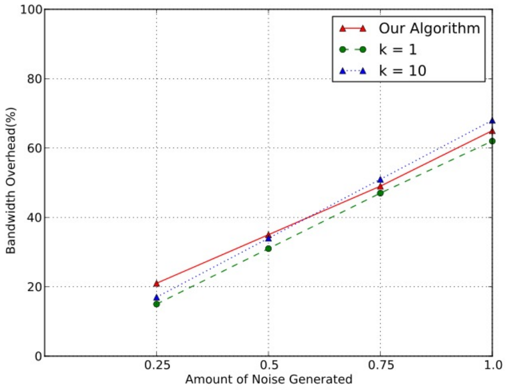


Figure 7. The bandwidth overhead when introducing different kinds of cover traffic.

Intuitively, accuracy should decrease as more noise is added. However, in Figure 6, we found that in our algorithm, $s = 0.25$ has a lower accuracy than $s = 0.5$. We hypothesized that this could be due to the features being considered. Recall that two of the five features are the total number of incoming packets and the total size of incoming packets. When s is lower, the number of incoming packets is lower. To verify our hypothesis, we re-ran our algorithms without considering these two features of incoming packets. Figure 8 shows the result. It can be seen that without these two features, accuracy decreases as noise generated increases, which is expected. This shows that the attributes for total size of incoming packets and total number of incoming packets help the website fingerprinting adversary in successfully identifying the correct website (increase in accuracy). Without these two features, our proposed algorithm performs even better as the accuracy is reduced to under 10% when $s = 1.0$. Previous work [19] has shown that the number of incoming packets is one of the most useful attributes in classification for website fingerprinting. We also considered not including the incoming packet features; the results are shown in Figure 9. The results are expected as well but the change in accuracy is not as obvious as Figure 8 since the number of outgoing packets is about the same regardless of the value of s.

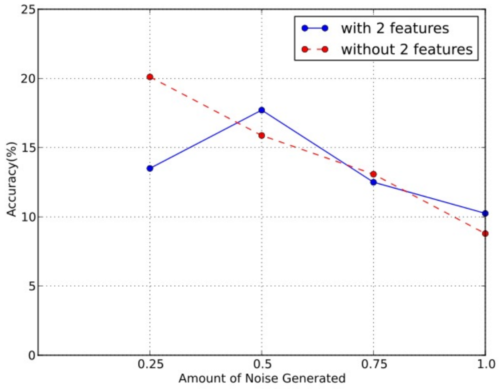


Figure 8. The accuracy using the Random Forest algorithm when considering the incoming packet features or not.

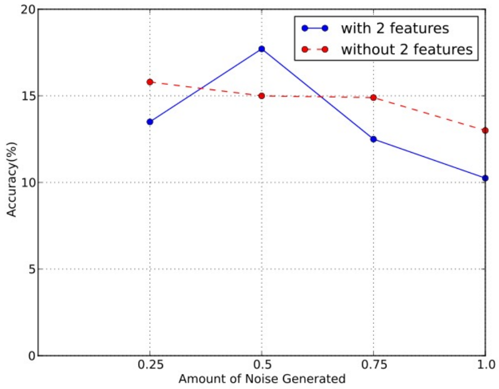


Figure 9. The accuracy using the Random Forest algorithm when considering the outgoing packet features or not.

6. Real-World Experiment

6.1. Setup

We collected network traffic data for the top 100 web sites listed from Wikipedia. After we removed duplicates (e.g. google.com and google.co.uk) and adult websites, we were left with 75 websites. For each website, we recorded 20 instances of network traffic through Tor without noise and 20 instances of network traffic through Tor including traffic from the noise server. Each network traffic trace's duration was two minutes. We used Wireshark version 1.6.7 to capture packets at the TCP level and TorBrowser version 6.5.2 as the web browser. The computers used were Dell Optiplex with Intel i5 and 4GB of RAM. We note that Wireshark cannot differentiate whether a packet is from the noise server or from the webserver. All the traffic looks like it originates from the Tor network. The following steps outline how we record the network traffic of a website.

1. Launch Wireshark to record network traffic
2. Launch TorBrowser and visit a webpage
3. Launch cover traffic client agent (only for the experiments with noise)
4. Wait 2 minutes
5. Save network traffic to file
6. Shut down Wireshark and TorBrowser

The cover traffic server is deployed on a different machine from the client. The cover traffic client agent runs Tor version 0.2.2.35; all the cover traffic are forwarded to the noise server through the Tor client. The noise traffic consists of the network traffic trace of the top 10 web pages; in reality, this would be the traffic trace of websites visited by the user. Each time the cover traffic client needs to generate noise, it will randomly pick on these ten traces and sends requests and receives responses from the noise server as described in Section 4.3. In our experiments, we set *MinTimediffBtwRequests* to 500 milliseconds, *MaxNumberOfRequest* to 10, *MaxNumberOfResponse* to 10 and *MinTimediffBtwResponses* to 200 milliseconds. Figure 1 illustrates how our experiments are set up.

6.2. Evaluation

We deployed our real-world experiments from May 13 to August 10. We visited 75 websites and collected data for 20 instances of each of the 75 websites. This was our training dataset. We then repeated the experiments for the same websites and number of instances for each website with cover traffic generated by the noise server. This made up our testing dataset. Similar to the simulations, we used the Random Forest classification algorithm to perform the website prediction. Figure 10 shows the accuracy of website fingerprinting. When no cover traffic is generated, the accuracy is 81.59%, which is close to the 90% obtained in previous

Mitigation	Accuracy (%)	Latency Overhead (%)	Bandwidth Overhead(%)
No Defense	91%	0%	0%
CS-BuFLO [7]	22%	173%	130%
Tamaraw [4]	10%	200%	38%
WTF-PAD [44]	15%	0%	54%
Walkie-Talkie [45]	19% ~ 44%	34%	31%
Our algorithm Simulation	14%	0%	20%
Our Real-world Experiment	16%	0%	20%

Table 4. Comparison of our algorithm's accuracy and overhead with previous mitigation schemes. We showed the lowest accuracy numbers for the other schemes, regardless of algorithms used. The table is based from [44].

research. However, when generating 10% cover traffic, the accuracy decreases to 17.81% which shows that our cover traffic algorithm is significantly impacting the adversary's ability to perform a website fingerprinting attack. When generating 20% and 30% cover traffic, the accuracy obtained is 16.37% and 10.72% respectively. We also ran the k-nearest neighbors classifier as this was used in previous research. The accuracy obtained is similar to that obtained when using the Random Forest algorithm.

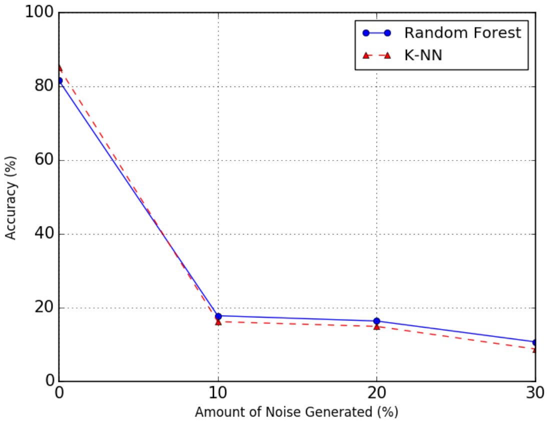


Figure 10. The accuracy of the website fingerprinting attack for real-world experiments with varying amounts of noise generated. Classifier used was Random Forest.

7. Discussion and Future Work

We showed that our proposed cover traffic (noise generation) algorithm mitigates website fingerprinting attacks as effectively as current existing schemes. However, the bandwidth overhead is only 20%, much lower than existing schemes. The latency overhead is also 0%. Our algorithm can also be configured to utilize different amounts of bandwidth, which affects the adversary's accuracy.

Recording user's browsing session: We emphasize that the user's webbrowsing session only need to be recorded locally. This information is not shared. Moreover, only the packet sizes and number of packets are recorded. The server and actual contents are not recorded. The information sent to the noise server is only the number of packets to send back and their size. The contents of the packets are random, not actual contents. Since the packets are encrypted, an adversary cannot determine that these packets are cover traffic. Since no actual contents are sent, our scheme does not leak any information to an eavesdropper.

Using a dedicated noise server: The cover traffic could be sent to real webservers; however, this would put extra strain on these servers. We, thus, decided to use a dedicated noise server. Removing the noise server will mean only outgoing cover traffic can be sent which could be filtered out by an adversary. Multiple noise servers, such as using Amazon cloud, could be used if this scheme is deployed. Since Tor is used, it cannot be determined whether the user is contacting a noise server.

We plan to expand this work in considering more webpages for both the training dataset and our learning algorithm. A more detailed study on the different classification algorithms and parameters used will also be performed. Further improvements to our algorithm can be made, such as, if a user has multiple tabs open at the same time, no noise is needed. This would reduce the bandwidth overhead.

Acknowledgment

The research results discussed in this publication were made possible in total or in part by funding through the Health Research award for project number HR13-035, from the Oklahoma Center for the Advancement of Science and Technology.

References

[1] Tor (2017), https://www.torproject.org/.

[2] Hintz, A. (2003) Fingerprinting websites using traffic analysis. In *Proceedings of the 2Nd International Conference on Privacy Enhancing Technologies*, PET'02 (Berlin, Heidelberg: Springer-Verlag): 171–178. URL `http://dl.acm.org/citation.cfm?id=1765299.1765312`.

[3] Cai, X., Zhang, X.C., Joshi, B. and Johnson, R. (2012) Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12 (New York, NY, USA: ACM): 605–616. doi:10.1145/2382196.2382260, URL `http://doi.acm.org/10.1145/2382196.2382260`.

[4] Wang, T., Cai, X., Nithyanand, R., Johnson, R. and Goldberg, I. (2014) Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14 (Berkeley, CA, USA: USENIX Association): 143–157. URL `http://dl.acm.org/citation.cfm?id=2671225.2671235`.

[5] Nithyanand, R., Cai, X. and Johnson, R. (2014) Glove: A bespoke website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14 (New York, NY, USA: ACM): 131–134. doi:10.1145/2665943.2665950, URL `http://doi.acm.org/10.1145/2665943.2665950`.

[6] Cai, X., Nithyanand, R., Wang, T., Johnson, R. and Goldberg, I. (2014) A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14 (New York, NY, USA: ACM): 227–238. doi:10.1145/2660267.2660362, URL `http://doi.acm.org/10.1145/2660267.2660362`.

[7] Cai, X., Nithyanand, R. and Johnson, R. (2014) Cs-buflo: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14 (New York, NY, USA: ACM): 121–130. doi:10.1145/2665943.2665949, URL `http://doi.acm.org/10.1145/2665943.2665949`.

[8] Cui, W., Yu, J., Gong, Y. and Chan-Tin, E. (2018) Realistic cover traffic to mitigate website fingerprinting attacks. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*: 1579–1584. doi:10.1109/ICDCS.2018.00175.

[9] Sun, Q., Simon, D.R., Wang, Y.M., Russell, W., Padmanabhan, V.N. and Qiu, L. (2002) Statistical identification of encrypted web browsing traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02 (Washington, DC, USA: IEEE Computer Society): 19–. URL `http://dl.acm.org/citation.cfm?id=829514.830535`.

[10] Bissias, G.D., Liberatore, M., Jensen, D. and Levine, B.N. (2006) Privacy vulnerabilities in encrypted http streams. In *Proceedings of the 5th International Conference on Privacy Enhancing Technologies*, PET'05 (Berlin, Heidelberg: Springer-Verlag): 1–11. doi:10.1007/11767831_1, URL `http://dx.doi.org/10.1007/11767831_1`.

[11] Liberatore, M. and Levine, B.N. (2006) Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06 (New York, NY, USA: ACM): 255–263. doi:10.1145/1180405.1180437, URL `http://doi.acm.org/10.1145/1180405.1180437`.

[12] Lu, L., Chang, E.C. and Chan, M.C. (2010) *Website Fingerprinting and Identification Using Ordered Feature Sequences* (Berlin, Heidelberg: Springer Berlin Heidelberg), 199–214. doi:10.1007/978-3-642-15497-3_13, URL `http://dx.doi.org/10.1007/978-3-642-15497-3_13`.

[13] Herrmann, D., Wendolsky, R. and Federrath, H. (2009) Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09 (New York, NY, USA: ACM): 31–42. doi:10.1145/1655008.1655013, URL `http://doi.acm.org/10.1145/1655008.1655013`.

[14] Panchenko, A., Niessen, L., Zinnen, A. and Engel, T. (2011) Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11 (New York, NY, USA: ACM): 103–114. doi:10.1145/2046556.2046570, URL `http://doi.acm.org/10.1145/2046556.2046570`.

[15] Gong, X., Borisov, N., Kiyavash, N. and Schear, N. (2012) Website detection using remote traffic analysis. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12 (Berlin, Heidelberg: Springer-Verlag): 58–78. doi:10.1007/978-3-642-31680-7_4, URL `http://dx.doi.org/10.1007/978-3-642-31680-7_4`.

[16] Wang, T. and Goldberg, I. (2013) Improved website fingerprinting on tor. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13 (New York, NY, USA: ACM): 201–212. doi:10.1145/2517840.2517851, URL `http://doi.acm.org/10.1145/2517840.2517851`.

[17] Juarez, M., Afroz, S., Acar, G., Diaz, C. and Greenstadt, R. (2014) A critical evaluation of website fingerprinting attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14 (New York, NY, USA: ACM): 263–274. doi:10.1145/2660267.2660368, URL `http://doi.acm.org/10.1145/2660267.2660368`.

[18] Panchenko, A., Lanze, F., Zinnen, A., Henze, M., Pennekamp, J., Wehrle, K. and Engel, T. (2016) Website fingerprinting at internet scale. In *Proceedings of the 23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*.

[19] Wang, T. and Goldberg, I. (2016) On realistically attacking tor with website fingerprinting. In *Privacy Enhancing Technologies Symposium (PETS)*.

[20] Spreitzer, R., Griesmayr, S., Korak, T. and Mangard, S. (2016) Exploiting data-usage statistics for website fingerprinting attacks on android. In *Proceedings of the 9th ACM Conference on Security &#38; Privacy in Wireless and Mobile Networks*, WiSec '16 (New York, NY, USA: ACM): 49–60. doi:10.1145/2939918.2939922, URL `http://doi.acm.org/10.1145/2939918.2939922`.

[21] Hayes, J. and Danezis, G. (2016) k-fingerprinting: A robust scalable website fingerprinting technique.

In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX: USENIX Association): 1187–1203. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes.

[22] Perry, M. (2011), Experimental defense for website traffic fingerprinting, https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting.

[23] Yu, J. and Chan-Tin, E. (2014) Identifying webbrowsers in encrypted communications. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14 (New York, NY, USA: ACM): 135–138. doi:10.1145/2665943.2665968, URL http://doi.acm.org/10.1145/2665943.2665968.

[24] Eckersley, P. (2010) How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10. URL http://dl.acm.org/citation.cfm?id=1881151.1881152.

[25] Miller, B., Huang, L., Joseph, A.D. and Tygar, J.D. (2014) *I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis* (Cham: Springer International Publishing), 143–163. doi:10.1007/978-3-319-08506-7_8, URL http://dx.doi.org/10.1007/978-3-319-08506-7_8.

[26] Le Blond, S., Choffnes, D., Zhou, W., Druschel, P., Ballani, H. and Francis, P. (2013) Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. doi:10.1145/2486001.2486002, URL http://doi.acm.org/10.1145/2486001.2486002.

[27] Dyer, K.P., Coull, S.E., Ristenpart, T. and Shrimpton, T. (2012) Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.

[28] Mittal, P., Khurshid, A., Juen, J., Caesar, M. and Borisov, N. (2011) Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS 2011)*.

[29] Tschantz, M.C., Afroz, S., Anonymous and Paxson, V. (2016) SoK: Towards Grounding Censorship Circumvention in Empiricism. *IEEE Symposium on Security and Privacy* .

[30] Wright, C., Coull, S. and Monrose, F. (2009) Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the Network and Distributed Security Symposium - NDSS '09* (IEEE).

[31] Houmansadr, A., Riedl, T.J., Borisov, N. and Singer, A.C. (2013) I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention. In *NDSS*.

[32] Moghaddam, H.M., Li, B., Derakhshani, M. and Goldberg, I. (2012) Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*.

[33] Fifield, D., Hardison, N., Ellithorpe, J., Stark, E., Boneh, D., Dingledine, R. and Porras, P. (2012) Evading censorship with browser-based proxies. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12 (Berlin, Heidelberg: Springer-Verlag): 239–258. doi:10.1007/978-3-642-31680-7_13, URL http://dx.doi.org/10.1007/978-3-642-31680-7_13.

[34] Wang, Q., Gong, X., Nguyen, G.T., Houmansadr, A. and Borisov, N. (2012) Censorspoofer: Asymmetric communication using ip spoofing for censorship-resistant web browsing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12 (New York, NY, USA: ACM): 121–132. doi:10.1145/2382196.2382212, URL http://doi.acm.org/10.1145/2382196.2382212.

[35] Holowczak, J. and Houmansadr, A. (2015) Cachebrowser: Bypassing chinese censorship without proxies using cached content. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (New York, NY, USA: ACM): 70–83. doi:10.1145/2810103.2813696, URL http://doi.acm.org/10.1145/2810103.2813696.

[36] Wang, L., Dyer, K.P., Akella, A., Ristenpart, T. and Shrimpton, T. (2015) Seeing through network-protocol obfuscation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (New York, NY, USA: ACM): 57–69. doi:10.1145/2810103.2813715, URL http://doi.acm.org/10.1145/2810103.2813715.

[37] Houmansadr, A., Brubaker, C. and Shmatikov, V. (2013) The parrot is dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13 (Washington, DC, USA: IEEE Computer Society): 65–79. doi:10.1109/SP.2013.14, URL http://dx.doi.org/10.1109/SP.2013.14.

[38] Geddes, J., Schuchard, M. and Hopper, N. (2013) Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13 (New York, NY, USA: ACM): 361–372. doi:10.1145/2508859.2516742, URL http://doi.acm.org/10.1145/2508859.2516742.

[39] Diaz, C. and Preneel, B. (2004) Taxonomy of mixes and dummy traffic. In *Proceedings of I-NetSec04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*.

[40] Mallesh, N. and Wright, M. (2007) Countering statistical disclosure with receiver-bound cover traffic. In Biskup, J. and Lopez, J. [eds.] *Proceedings of 12th European Symposium On Research In Computer Security (ESORICS 2007)* (Springer), *Lecture Notes in Computer Science* **4734**: 547–562.

[41] Simon Oya, C.T. and Pérez-González, F. (2014) Do dummies pay off? limits of dummy traffic protection in anonymous communications. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium (PETS 2014)*.

[42] Howe, D. and Nissenbaum, H. (2008) Trackmenot: Resisting surveillance in web search. *On the Identity Trail: Privacy, Anonymity and Identify in a Networked*

Society .

[43] Peddinti, S. and Saxena, N. (2010) On the privacy of web search based on query obfuscation: A case study of trackmenot. In Atallah, M. and Hopper, N. [eds.] *Privacy Enhancing Technologies* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **6205**, 19–37. doi:10.1007/978-3-642-14527-8_2, URL http://dx.doi.org/10.1007/978-3-642-14527-8_2.

[44] Juarez, M., Imani, M., Perry, M., Diaz, C. and Wright, M. (2016) Toward an efficient website fingerprinting defense. In *ESORICS*.

[45] Wang, T. and Goldberg, I. (2017) Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC: USENIX Association): 1375–1390. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tao.

[46] Wright, C., Coull, S. and Monrose, F. (2009) Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the Network and Distributed Security Symposium - NDSS '09* (IEEE).

[47] Weinberg, Z., Wang, J., Yegneswaran, V., Briesemeister, L., Cheung, S., Wang, F. and Boneh, D. (2012) StegoTorus: A camouflage proxy for the Tor anonymity system. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*.

[48] Dingledine, R., Mathewson, N. and Syverson, P. (2004) Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*.

[49] Weka (2017), http://www.cs.waikato.ac.nz/ml/weka/.