Giovanni Cherubin*, Jamie Hayes*, and Marc Juarez*

Website Fingerprinting Defenses at the Application Layer

Abstract: *Website Fingerprinting* (WF) allows a passive network adversary to learn the websites that a client visits by analyzing traffic patterns that are unique to each website. It has been recently shown that these attacks are particularly effective against `.onion` sites, anonymous web servers hosted within the Tor network. Given the sensitive nature of the content of these services, the implications of WF on the Tor network are alarming. Prior work has only considered defenses at the client-side arguing that web servers lack of incentives to adopt countermeasures. Furthermore, most of these defenses have been designed to operate on the stream of network packets, making practical deployment difficult. In this paper, we propose two application-level defenses including the first server-side defense against WF, as `.onion` services have incentives to support it. The other defense is a lightweight client-side defense implemented as a browser add-on, improving ease of deployment over previous approaches. In our evaluations, the server-side defense is able to reduce WF accuracy on Tor `.onion` sites from 69.6% to 10% and the client-side defense reduces accuracy from 64% to 31.5%.

Keywords: privacy, anonymity, website fingerprinting

DOI 10.1515/popets-2017-0023
Received 2016-08-31; revised 2016-11-30; accepted 2016-12-01.

1 Introduction

Website Fingerprinting (WF) attacks allow a passive local adversary to infer which webpage a client is viewing by identifying patterns in network traffic that are unique to the webpage. These attacks are possible even if the client is browsing through anonymity networks such as Tor and the communication is encrypted [12]. Tor routes a client's traffic through volunteer relays before connecting to the communication's destination, so that local eavesdroppers cannot link both sender and receiver of the communication [8]. However, the WF attack, if successful, breaks the *unlinkability* property that Tor aims to provide to its users.

Moreover, a 2015 study has shown that `.onion` sites can be distinguished from regular sites with more than 90% accuracy [16]. This substantially narrows down the classification space in Tor and suggests the attack is potentially more effective at identifying `.onion` sites than regular pages. *Onion services* are websites with the `.onion` domain hosted over Tor, allowing a client to visit a website without requiring it to publicly announce its IP address. These sites tend to host sensitive content and may be more interesting for an adversary, turning the WF attack into a major threat for connecting users. In this paper, we propose the first set of defenses specifically designed and evaluated for Tor `.onion` sites.

WF defenses are often theorized at the network level, and try to disrupt statistical patterns via inserting dummy messages in to the packet stream [2, 4, 9]. Some defenses try to alter the network traffic of a webpage to mimic that of another webpage that is not interesting to the attacker [32]. However, a defense at the network level may require substantial changes of Tor or even the TCP stack, which would make its deployment unrealistic. Furthermore, there is no need to hide patterns at the network layer because few webpage-identifying features, if any, are introduced by low layers of the stack (e.g., TCP, IP). In this work, we consider application-layer defenses, arguing that this approach is more natural for WF defenses and facilitates their development.

Existing WF defenses have been engineered to protect the link between the client and the entry to the Tor network, assuming this is the only part of the network observable by the adversary. We propose both WF defenses at the client- and server-side. A server-side defense is more usable as it does not require any action from the user. More and more, certain types of websites, such as human rights advocacy websites, have the motivation to provide WF defenses as a service to its user base, who may be of particular interest to an adversary. For this reason, we believe that, in contrast to *normal* websites, `.onion` site operators not only have the incentive to provide defenses against WF attacks, but can also achieve a competitive advantage with respect to other `.onion` sites by doing so.

*All are corresponding authors and share first authorship.
Giovanni Cherubin: Royal Holloway University of London, Egham, UK, E-mail: Giovanni.Cherubin.2013@live.rhul.ac.uk
Jamie Hayes: University College London, London, UK, E-mail: j.hayes@cs.ucl.ac.uk
Marc Juarez: imec-COSIC KU Leuven, Leuven, Belgium, E-mail: marc.juarez@kuleuven.be

As a real life motivating example, we were contacted by *SecureDrop* [27], an organization that provides onion services for the anonymous communication between journalists and whistleblowers. They are concerned that sources wishing to use their service can be de-anonymized through WF. As a consequence, they are interested in using a server-side WF defense. We have included a SecureDrop website in all the datasets used for the evaluation of defenses.

We introduce two variants of a server-side defense operating at the application layer, which we call *Application Layer Padding Concerns Adversaries* (ALPaCA). We evaluate it via a live implementation on the Tor network. We first crawl over a significant fraction of the total Tor `.onion` site space, retrieving not only the network level traffic information – as is standard in WF research – but also the `index.html` page and HTTP requests and responses. We then analyze the size distribution for each content type, e.g. PNG, HTML, CSS. Using this information, ALPaCA alters the `index.html` of a page to conform to an "average" `.onion` site page. ALPaCA runs periodically, changing the page fingerprint on every user request.

Due to the expected slow adoption of server-side WF defenses, client-side defenses must still be used. We therefore implement a simple client-side WF defense, dubbed *Lightweight application-Layer Masquerading Add-on* (LLaMA), that works at the application layer by adding extra delays to the HTTP requests. These delays alter the order of the requests in a similar way to randomized pipelining (RP) [23], a WF countermeasure implemented in the Tor browser that has been shown to fail in several evaluations [5, 14, 31]. Besides delaying HTTP requests, our defense sends redundant requests to the server. We show most of the protection provided by this defense stems from the extra requests and not from the randomization of legitimate requests.

Our contributions are, as a result of a real life demand, the first implementation of a server-side WF defense and a simple yet effective lightweight client-side defense. With these two approaches we explore the space of application-layer defenses specifically designed to counter WF in `.onion` sites. In addition, we have collected the largest – to the best of our knowledge – dataset of sizes and types of content hosted by Tor `.onion` sites. We provide an evaluation of the overhead and efficacy of our defenses and compare it to some of the most practicable existing WF defenses.

The source code and datasets of both ALPaCA and LLaMA have been made publicly available on GitHub[1]. The original code is also available on an `.onion` site[2], which is protected using our defense.

2 Threat Model

As depicted in Figure 1, we consider an adversary who has access to the communication between the client and the entry point to the Tor network, known as *entry guard*. A wide range of actors could have access to such communications, ranging from malicious or corrupted relay operators, who can target all clients connecting to the guards they control; to ASes, ISPs and local network administrators, who can eavesdrop on Tor clients located within their infrastructure.

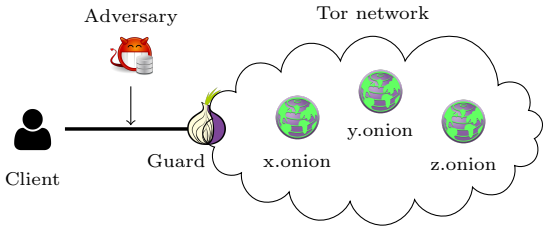

Fig. 1. A client visits an `.onion` site over Tor. The attacker eavesdrops the encrypted link between the Tor client and the entry *guard* to the Tor network. Between the client and the destination onion service there is a six-hop Tor circuit that we have omitted to simplify the figure.

The adversary eavesdrops the communication to obtain a *trace* or *sample instance* of the encrypted network packets. He can observe and record these packets, but he cannot decrypt them. Furthermore, we will assume a *passive* adversary: he cannot remove or modify the packets, nor drop or add new packets to the stream.

The objective of the adversary is to infer the websites that were visited by the client from the traffic samples. The adversary can build a template for a number of websites with his own visits and then match the traffic generated by the client. It has been shown that, for a small set of websites, such an attacker can achieve high success rates achieving over 90% accuracy [5].

1 http://github.com/camelids/
2 http://3tmaadslguc72xc2.onion

These attacks have however been criticized for making a number of unrealistic assumptions that favor the adversary [14]. For instance, they assume webpages are static, although some pages have frequent content updates; the client only visits pages that the attacker has trained on, also known as the *closed-world* assumption; and the attacker is able to perfectly parse the fraction of the continuous stream of traffic corresponding to a specific page download, assuming there is a gap between one visit and the next one.

In 2015, Kwon et al. showed that an attacker falling within this threat model can effectively distinguish visits to `.onion` sites from regular websites [16]. They also revisited the assumptions for which prior work on WF had been criticized [14] and found that many of these assumptions hold when considering only `.onion` sites. In contrast to the open Web, the world of `.onion` sites is small and comparable to a closed world, they are also more static than regular websites and their streams are isolated by domain [16]. As in virtually all prior work on WF, they still assumed the client visits only home pages, ignoring other pages in the website such as inner pages and logged-in or personalized pages that are not available to the attacker. In our evaluation, we follow them and only collect data for the home page of the `.onion` sites we have crawled.

We assume an adversary is only interested in fingerprinting `.onion` sites, and already has a classifier to tell `.onion` traffic apart from the bulk of client traffic. We focus on defenses that protect against the WF attack in the "onion world" because it is a more threatening setting than the one studied in most prior WF work on Tor; visits to `.onion` sites tend to be more sensitive than to pages whose IP address is visible to clients. Luo et al. argue that a WF defense must be implemented at the client-side because web servers have no incentive to offer such a service [19]. However, we believe `.onion` site operators are aware of the privacy concerns that Tor clients have and would make the necessary (minor) modifications in the server to implement our defense.

For the design of ALPaCA, we will assume there is no dynamic content. This includes content generated at the client-side (e.g., AJAX) as well as the server-side (e.g., a PHP script polling a database). This assumption simplifies the design of the server-side defense: ALPaCA requires the size of the web resources being loaded and it is hard to estimate the size of dynamic content a priori.

To assume that no JavaScript will run in the browser is not as unrealistic as it may seem given the high prevalence of JavaScript in the modern Web. The Tor Browser's security slider allows users to select different levels of security, disabling partially or totally JavaScript. Furthermore, SecureDrop pages already ask their clients to disable JavaScript to prevent attacks such as cross-site scripting. It is reasonable to think that clients who protect themselves against WF will first disable JavaScript to prevent these other attacks.

3 Related Work

WF is typically modeled as a supervised learning problem. The attacker collects traffic traces for a large sample of websites that aims to identify and builds a classifier that outputs a label, with a certain level of confidence. Since the first WF classifiers were proposed in the late nineties [7], the attacks have been developed with improved classification models to defeat a wide variety of privacy enhancing technologies such as encrypting web proxies [13, 28], SSH tunnels[17], VPNs, and even anonymity systems such as Tor and JAP [12].

3.1 Attacks

The latest attacks against Tor achieve more than 90% accuracy in a *closed-world* of websites, where the attacker is assumed to have samples for all the websites a target user may visit [5, 11, 20, 30, 31]. This assumption is unrealistically advantageous for the attacker [14] and a recent study has shown that the attack does not scale to large open-worlds [20]. However, the `.onion` space is significantly smaller than the Web and may be feasible for an adversary to train on a substantial fraction of all `.onion` websites. Furthermore, the closed-world evaluation provides a lower bound for the efficacy of the defense. For a complete evaluation of the performance of our defenses, in this paper we will provide results for both open and closed-world scenarios.

We have selected the most relevant attacks in the literature to evaluate our defenses:

k-NN [30]: Wang et al. proposed a feature set of more than 3,000 traffic features and defined an adaptive distance that gives more weight to those features that provide more information. To classify a new instance, the attack takes the label of the k Nearest Neighbors (k-NN) and only makes a guess if all the neighbors agree, minimizing the number of false positives.

CUMUL [20]: The features of this attack are based on the cumulative sums of packet sizes. The authors interpolated a fixed number of points from this cumulative sum to build the feature vectors that they use to feed a Support Vector Machine (SVM).

k-FP [11]: Hayes and Danezis used Random Forests (RF) to transform, based on the leafs of the trees, an initial feature set to another feature set that encodes the similarity of an instance with respect to its neighbors. Then, they also used a k-NN for final classification.

3.2 Defenses

Most WF defenses in the literature are based on *link-padding*. The traffic morphing approach attempts to transform the traffic of a page to resemble that of another page [18, 21, 32], or to generalize groups of traffic traces in to anonymity sets [3, 30]. The main downside of this type of defenses is that they require a large database of traffic traces that would be costly to maintain [14].

Link-padding aims to conceal patterns in web traffic by adding varying amounts of dummy messages and delays in flows. Link-padding has been used for traffic morphing to cause confusion in the classifier by disguising the target page fingerprint as that of another page [32]. However, as Dyer et al. note [9], traffic morphing techniques produce high bandwidth overheads as some packets must be buffered for a long period. The strategy we follow in ALPaCA is different from traffic morphing in that page contents are not disguised as other pages', but rather the content is modified to become less *fingerprintable*. The intuition behind ALPaCA is to make each resource look like an "average" resource, according to the distribution of resources in the world of pages. This approach reduces the overheads with respect to morphing, as resizing an object to an average size will tend to require less amount of padding than to an object of a specific page. We have experimented with morphing the contents in a page to make it look like another page. This can be seen as the application-level counterpart of *traffic morphing* and the results can be found in Appendix A.

In 2012, Dyer et al. presented *BuFLO* [9], a defense based on constant-rate link-padding. Although *BuFLO* is a proof-of-concept defense and has high bandwidth overheads, other defenses have been developed from the original *BuFLO* design. *Tamaraw* [4] and *CS-BuFLO* [2] optimize *BuFLO*'s bandwidth and latency overheads to make its deployment feasible. Both of these defenses address the issue of padding the page's tail. BuFLO padded all pages up to a certain maximum number of bytes producing the high bandwidth overheads. CS-BuFLO and Tamaraw proposed a strategy to pad pages to multiples of a certain parameter, which groups pages in anonymity sets by size and significantly reduces the bandwidth overhead over BuFLO. We follow a similar strategy for one of the modes of ALPaCA.

Recently, a lightweight defense based on Adaptive Padding has also been proposed to counter WF [15]. In order to offer low latency overheads, this defense only pads time gaps in traffic that are statistically unlikely to happen. To empirically determine the likelihood of a gap they sampled a large number of pages over Tor and built a distribution of inter-arrival times used to sample the delays for the dummy messages.

Our main concern with these designs is that padding is applied at the network layer. There is no need to apply the defense at the network layer because layers below HTTP do not carry identifying information about the webpage. One could argue that latency and bandwidth identify the web server. However, these features vary depending on network conditions and are shared by all pages hosted in the same server or behind the same CDN. Moreover, the implementation of such defenses may require modifications in the Tor protocol and even the TCP stack, as they generate Tor cells that are sent over Tor's TLS connections.

Application layer defenses act directly on the objects that are fingerprinted at the network layer. The padding is also added directly to these objects. As opposed to network-layer defenses that must model legitimate traffic to generate padding, application-layer defenses inject the padding inside the encrypted payload and is, consequently, already indistinguishable from legitimate traffic at the network layer. In addition, defenses at the application layer do not require modifications in the source code of the Tor protocol, which make them more suitable for deployment.

In this paper we present and explore two novel approaches for application layer defenses at both client and server-side. In the rest of this section we describe the state of the art on application-layer defenses.

3.2.1 Server-side

To the best of our knowledge, there is only a prototype of a server-side defense that was drafted by Chen et al. and it was designed for a slightly different although related problem [6]. They studied WF in the context of SSL web applications, where the attacker is not trying

to fingerprint websites, but specific pages within one single website. Their main contribution was to show that a local passive adversary can identify fine-grained user interactions within the website. The authors devise a defense that requires modifications at both client and server sides, which allows padding to be added to individual HTTP requests and responses.

3.2.2 Client-side

There are only two application-layer defenses proposed in the WF literature: *HTTPOS* [19] and *Randomized Pipelining* (RP) [23]. Luo et al. proposed *HTTPOS* as a client-side defense arguing that server-side or hybrid defenses would see little adoption in the wild due to lack of incentives [19]. In that study, the authors pinpoint a number of high-level techniques that alter the traffic features exploited by WF attacks. For instance, they modify the HTTP headers and inject fake HTTP requests to modify the length of web object sizes.

RP is the only WF countermeasure that is currently implemented in the Tor browser. It operates by batching together a single clients requests in the HTTP pipeline to randomize their order before being sent to the server. Several studies have applied WF attacks on data collected with a RP-enabled Tor Browser and all of them have shown that the defense was not effective at decreasing the accuracy of the WF attack in the closed world [5, 14, 31]. The reason why RP does not work is not clear and has not been investigated in these evaluations.

4 Defenses

WF attacks are possible because different webpages serve different content. High level features such as the number of requests the browser makes to download a page, the order of these requests and the size of each response, induce distinctive low level features observed in the network traffic [9, 21]. For instance, the number of requests sent by the browser corresponds to the number of objects embedded in the page such as images, scripts, stylesheets, and so on.

Most existing defenses propose to add spurious network packets to the stream to hide these low-level features [2, 4, 9], However, effectively concealing these features at network level poses technical challenges, as the operation of underlying protocols, i.e. TLS, TCP, IP, obfuscates the relation between low and high level features. For this reason, we believe adding the padding to the actual contents of the page is a more natural strategy to hide traffic features than sending dummy packets: if the defense successfully conceals high-level features, the low-level features will follow.

In this section, we describe in detail the strategies that we propose at the application layer at both server (ALPaCA) and client side (LLaMA) to mitigate WF attacks.

4.1 ALPaCA

ALPaCA is a server-side defense that pads the contents of a webpage and creates new content with the objective of concealing distinctive features at the network level. We demonstrate that this strategy is not only effective, but also practical to deploy. We have implemented and evaluated ALPaCA as a script that periodically runs on a server hosting an `.onion` site.

We first show that it is possible to pad the most popular types of webpage objects (e.g., images, HTML) to a desired size, without altering how they look to a user. We then propose two variants of server-side defenses, referred to as P-ALPaCA and D-ALPaCA. At a high level, the defenses choose, for a page to morph, a suitable list of sizes T, that we call *target*. A target specifies the number and size of the objects of the morphed page; P-ALPaCA and D-ALPaCA differ in how they select such a target. Then, the objects of the original page are padded to match the sizes defined in T. If T contains more elements than the page's objects, then new objects ("padding objects") are created and referenced from the morphed HTML page (Algorithm 1). Figure 2 gives a high level overview of this process.

4.1.1 Padding an object to a target size

This section describes how we can pad most types of objects. It is important to note that an adversary looking at encrypted packets cannot: i) distinguish the type of objects that are being downloaded, ii) infer how much padding was added to such objects or whether they were padded at all. By padding an object directly on the server, we can control how large it will look like at the network level. While this control is not complete (because of compression in the HTTP protocol), experiments show that this discrepancy does not largely affect on our defenses.

Table 1. Padding the most frequent objects in `.onion` sites to a desired size. "N.O." stands for "not observed". We assume JavaScript is disabled, although it is possible to morph JS files as shown.

Content type	Morphing	Frequency
PNG, ICO, JPG, GIF, BMP	Append random bytes to the file.	51%
HTML	Insert random data within a comment "<!–", "–>".	13%
CSS	Insert random data within a comment "/*" "*/".	12%
JS	Insert random data within a comment "/*" "*/".	13%
MP3	Append random bytes to the file.	N.O.
AVI	Append random bytes to the file.	N.O.

Table 1 shows the types of objects that we can pad up to a desired size, and their frequency within the `.onion` site world. To pad text objects (e.g., HTML and CCS) we can add the desired amount of random data into a comment. To pad binary objects (e.g., images), it is normally sufficient to append random data to the end of the file; in fact, the file structure allows programs to recognize the end of the file even after this operation.

We verified that binary files would not be corrupted after appending random bytes to them as follows. We used ImageMagick's `identify` program[3] for verifying the validity of PNG, ICO, JPEG, GIF, and BMP files after morphing. The program only raised a warning "length and filesize do not match" for the BMP file; the image was, nevertheless, unaffected, as it could be opened without any errors. We used `mp3val`[4] to check MP3 files; the program returned a warning "Garbage at the end of the file", but the file was not corrupted, and it could be played. We used `ffmpeg`[5] to verify AVI files; the program did not return any errors or warnings.

It is thus possible to morph the most common object types. We suspect that many other types of object can be morphed analogously, by appending random bytes or by inserting data in comments or unused sections of the type structure. We remark, however, that in experiments we did not remove content we could not morph from webpages.

4.1.2 Morphing a page to a target T

We introduce Algorithm 1, which morphs the contents of a page to match the sizes defined by a target T. The target is selected differently by the two versions of ALPaCA, as presented later, and it defines the size of the objects that the morphed page should have.

The algorithm keeps two lists: M, containing the morphed objects, and P, which keeps track of the sizes in T that have not been used for moprhing an object; both lists M and P are initially empty. The algorithm sequentially considers the objects of the original page from the smallest to the largest; for object o, it seeks the smallest size $t \in T$ which o can be padded (i.e., for which $size(o) \leq t$). Once it has found such a t, it removes all the elements of T smaller than t, and pads o to size t; the elements removed from T at this stage (except t) are put into P. After all the original objects have been morphed, the sizes remaining in T are appended to P. New "padding objects" (objects containing random bytes) are generated according to the sizes in P. We make sure that padding objects will be downloaded by a browser, but will not be shown, by inserting a reference to them in the HTML page as if they were hidden images[6]. Finally, the HTML page itself is padded to a target size by the defense.

4.1.3 P-ALPaCA

P-ALPaCA (Probabilistic-ALPaCA) generates a target by randomly sampling from a distribution that represents real-world `.onion` sites. Specifically, it has access to three probability distributions D_n, D_h and D_s, defined respectively on the number of objects a page has, the size of the HTML page and the size of each of its objects. The defense samples a target T using these distributions, and then morphs the original page as shown in Algorithm 1.

We estimated D_n, D_h and D_s using Kernel Density Estimation (KDE) from $5,295$ unique `.onion` websites we crawled. Details about crawling and analysis of these websites are in section 5. In Appendix B we show the

3 http://www.imagemagick.org/

4 http://mp3val.sourceforge.net/

5 https://ffmpeg.org/

6 To add an invisible object called "rnd.png" to an HTML page we insert `<img src="rnd.png" style="visibility:hidden">'`. The browser will consider this a PNG file and it will download it, but it will not attempt to show it. The file, thus, needs not to respect the PNG format, and it can just contain random bytes.

Algorithm 1 Pad a list of objects to a target

Input: O: list of original objects
T: list of target sizes
Output: M: list of morphed objects

$M \leftarrow [\,]$
$P \leftarrow [\,]$
▷ Morph the original objects.
while $|M| < |O|$ **do**
  $o \leftarrow \arg\min_{o \in O} size(o)$
  ▷ Remove the target sizes smaller than $size(o)$.
  while $min(T) < size(o)$ **do**
    Remove $min(T)$ from T
    Append $min(T)$ to P
  end while
  if T is empty **then**
    ▷ Cannot morph O to T
    fail
  end if
  ▷ Note: the current $min(T)$ is larger than $size(o)$
  $t \leftarrow min(T)$
  $m \leftarrow o$ padded to size t
  Append m to M
end while
▷ Add padding objects.
Merge P and T into P
for p in P **do**
  $m \leftarrow$ New padding object of size p
  Append m to M
end for

resulting distributions D_n, D_h and D_s, and provide details on how we used KDE to estimate them.

The defense first samples the number of objects n for the morphed page according to D_n. Then, it samples the size of the morphed HTML from D_h, and n sizes from D_s which constitute a target T. Finally, it attempts to morph the original page to T (Algorithm 1); if morphing fails, the procedure is repeated. The algorithm is shown in Algorithm 2.

Because sampling from the distributions can (with low probability) produce very large targets T, we introduced a parameter *max_bandwidth* to P-ALPaCA. Before morphing, the defense checks that the total page size is smaller than or equal to this parameter: $\sum_{t \in T} t \leq max\_bandwidth$. If not, the sampling procedure is repeated.

A simple alternative to sampling from a distribution that represents the present state of the `.onion` world, is to sample the number and size of padding objects uniformly at random. We expect that this alternative approach would also set a maximum bandwidth parameter, which would serve as the upper bound of the size of the morphed page. One could imagine that a naive implementation of this alternative approach which sets a high maximum would cause extremely high bandwidth overheads. However, reducing this maximum parameter would constrain the morphed page to look like a small subsection of the onion world, removing altogether the possibility that the page is morphed to resemble a large `.onion` site. Our approach allows a large maximum bandwidth parameter to bet set while ensuring bandwidth overheads will be low. With our approach, the probability that a small page, say `A.onion`, is morphed to the size of a large `.onion` site, say `B.onion`, directly corresponds to the ratio of the number of `.onion` sites within the `.onion` world that are of an equal size to `B.onion`. Meaning a small `.onion` site can have the entire `.onion` world as an anonymity set while ensuring a low bandwidth overhead.

Algorithm 2 P-ALPaCA

Input: O: list of original objects
D_n: distribution over the number of objects
D_h: distribution over the size of HTML pages
D_s: distribution over the size of objects
html_size: size of the original HTML page
max_bandwidth: maximum page size

▷ We use $x \leftarrow^{\$} D$ to indicate that x is sampled from distribution D
$morphed \leftarrow False$
while not *morphed* **do**
  $T \leftarrow [\,]$
  $h \leftarrow^{\$} D_h$
  if $h < html\_size$ **then**
    continue
  end if
  $n \leftarrow^{\$} D_n$
  for i in $1..n$ **do**
    $s \leftarrow^{\$} D_s$
    Append s to T
  end for
  if $sum(T) < max\_bandwidth$ **then**
    Try morphing O to target T (Algorithm 1)
    If successful, $morphed \leftarrow True$
  end if
end while
Pad the HTML page to size h

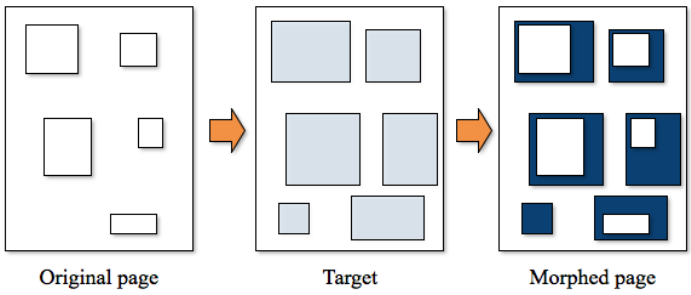

Fig. 2. Graphical representation of the server side defenses. Server side defenses P-ALPaCA and D-ALPaCA first select a target for the original web page. Then, they pad the contents of the original page as defined by the target (Algorithm 1), and generate new padding objects if needed. The original and morphed page will look identical to a user.

4.1.4 D-ALPaCA

We propose a second server-side defense, D-ALPaCA (Deterministic-ALPaCA), which decides deterministically by how much a page's objects should be padded. The defense is inspired by Tamaraw [4], which pads the number of packets in a network trace to a multiple of a padding parameter L. D-ALPaCA has the advantage of introducing less overheads than P-ALPaCA, but experimental results suggest this defense is slightly less effective against a WF adversary.

D-ALPaCA (Algorithm 3) accepts as input three parameters: λ, σ and max_s, where max_s should be a multiple of σ. It pads the number of objects of a page to the next multiple of λ, and the size of each object to the next multiple of σ. Then, if the target number of objects is larger than the original number of objects, it creates padding objects of size sampled uniformly at random from $\{\sigma, 2\sigma, ..., max\_s\}$. Experiments in section 6 evaluate how different sets of parameters influence security and overheads.

Algorithm 3 D-ALPaCA

Input: O: list of original objects
σ: size parameter
λ: number of objects parameter
$html\_size$: size of the original HTML page
max_s: maximum size of a padding object (should be a multiple of σ)

▷ We use $x \leftarrow^{\$} S$ to indicate that x is sampled uniformly at random from a set S
$T \leftarrow [\,]$
$h \leftarrow$ next multiple of σ greater or equal to $html\_size$
for o in O **do**
  $s \leftarrow$ next multiple of σ greater or equal to $size(o)$
  Append s to T
end for
$n \leftarrow$ next multiple of λ greater or equal to $size(O)$
while $size(T) < n$ **do**
  $s \leftarrow^{\$} \{\sigma, 2\sigma, ..., max\_s\}$
  Append s to T
end while
Morph O to target T (Algorithm 1)
Pad the HTML page to size h

4.1.5 Practicality of the defenses

Both P-ALPaCA and D-ALPaCA are practical to use in real-world applications. In fact, they only require a script to morph the contents of a page periodically. This can be done by setting up a `cron` job running the defense's code, which we release.

Since it is preferable to morph a page after each client's visit, and it may be difficult for the server operator to decide how frequently they should run the `cron` job, we propose a more sophisticated (and flexible) alternative. The defense should preemptively morph the web page many times, and place the morphed pages within distinct directories on the server. Then, the server should be configured to redirect every new request to a different directory. Once the content of a directory has been loaded, the directory is removed, and a new one can be created.

4.1.6 Third-party content

A limitation of ALPaCA is that it can only pad resources hosted in the web server, thus content linked from third parties cannot be protected. In the evaluation

of the defense, we have intentionally omitted all third-party content because only two out of the 100 pages in our dataset had resources from third parties.

To understand the impact of this assumption on a larger scale, we have analyzed the prevalence of third-party resources in a crawl of 25K `.onion` sites: only 20% of these sites create requests to third-party domains. Furthermore, for half the pages with third-party content, the third-party requests account for less than 40% of total requests observed within a webpage. However, we found a handful of sites that had more than 90% of their content hosted in third parties. They seem to act as proxies to existing websites. With such a high percentage of unprotected content, the defense is most likely to fail at providing protection against website fingerprinting.

Since the average cost in terms of disk space is 5MB, a possible solution for sites with a large proportion of third-party content would be to cache the third-party resources in the server running the defense. We strongly discourage this approach as if not implemented properly, the `.onion` site, attempting to keep these resources updated, may become vulnerable to timing correlations attacks by the third parties serving the content. In fact, we recommend `.onion` site operators minimize the amount of third-party content they embed to their pages and only cache static content that does not require periodic updates.

4.2 LLaMA

LLaMA is inspired by Randomized Pipelining (RP) [23]. RP modifies the implementation of HTTP pipelining in Firefox to randomize the order of the HTTP requests queued in the pipeline. However, RP has been shown to fail at thwarting WF attacks in several evaluations [5, 14, 31].

LLaMA is implemented as an add-on for the Tor browser that follows a similar strategy to RP: it alters the order in which HTTP requests are sent. The main advantage of a WF defense as a browser add-on is ease of deployment: it does not require modifications to the Tor source code. Thus, a user can install the add-on to enable the protection offered by the defense independently or, if the Tor Project decides to, it could be shipped with the Tor Browser Bundle.

RP exposes a debug flag that logs extra information about its use of the HTTP pipeline [22]. A dataset collected with this flag enabled, visiting the same webpages that the aforementioned evaluations did, provided evidence of a suboptimal usage of the HTTP pipeline by RP [24]. Either the design of those pages or the low adoption of HTTP pipelining on the servers of these pages or CDNs in between may account for the low performance of RP [1]. Since our defense does not depend on HTTP pipelining, it allows us to test whether these hypotheses hold or it is actually the randomization strategy which is flawed.

Delaying requests. In order to randomize the order of the HTTP requests, the add-on intercepts all requests generated during a visit to a website and adds a different random delay to each one (see Figure 3). We use the statistics extracted from subsection 5.2 to set the distribution of delays for the requests. We take the median page load time in our crawl and set a uniform distribution from zero to half the median load time. As a result, on average, each request will be delayed within a window of half the page load time. In the worst case, this approach will introduce 50% latency overhead if the last request is delayed by the maximum time in the distribution.

Extra requests. As shown in Figure 3, every time a request is sent or a response is received, the extension can be configured to send an extra request. It tosses a coin to decide whether to make an additional HTTP request or not. These fake HTTP requests are sent to a web server that serves custom-sized resources: a parameter in the URL indicates the size of the resource that will be sent in the response body. This allows us to fake random responses from the client-side. Tor isolates streams in different circuits per domain, since such fake requests are made to a different domain they will be sent through a different circuit. This should not be a problem because the attacker cannot distinguish them from legitimate third-party requests. However, as we discuss in the following section, third-party content in `.onion` sites has low prevalence. In addition, this approach requires a trusted server that can be queried from the add-ons. To avoid these issues, the extension implements an alternative method to generate extra responses: it keeps a hash table with domains as keys and lists of request URLs sent to that domain during a browser session as values. To generate a new request, it uniformly samples a URL from the list corresponding to the current first-party domain and sends a request to that URL.

To change the size of legitimate requests we would require cooperation of the server. We acknowledge that previous defenses have proposed this approach [6], but our focus for this defense is to not require any change at the server-side.

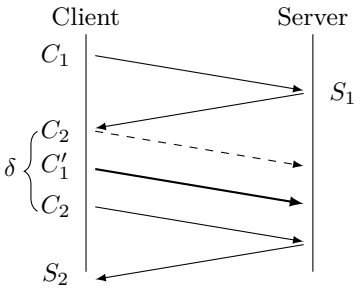

Fig. 3. Graphical representation of the LLaMA's operation. δ is the delay added to C_2. C'_1, in bold, requests the same resource as C_1.

5 Methodology

In this section we describe the methodology that we followed to collect the data and evaluate the defenses. This data was also used to create the probability distribution used by P-ALPaCA.

5.1 Data collection

For the collection of the dataset we used the `tor--browser-crawler`[7], a web crawler that provides a driver for the Tor Browser, allowing the automation of web page visits in conditions similar to those of regular Tor users. We added support for the Tor Browser Bundle 5.5.5, the latest version at the time of our crawls (March 2016) and extended the crawler to intercept all HTTP requests and responses for future inspection. The crawler logs the size and the URL for each HTTP request and response. The crawler also allows to modify browser preferences. We used this feature to disable JavaScript and RP when needed.

We crawled a list of `.onion` sites obtained from *Ahmia*[8], the most popular search engine for onion services. Ahmia maintains a blacklist of illegal `.onion` sites and thus are excluded from our crawls. The crawl consisted of 25,000 `.onion` instances, after removing timeouts and failed loads, we captured 18,261 instances of an `.onion` site load from 5,295 unique addresses. This dataset serves as both the basis for which we conduct WF attack experiments with our defense in place, as a source of information when inferring the distribution of objects that the server-side defense should conform to, and as a source of load time statistics for which the client-side defense decides when to inject additional requests.

5.2 Data analysis

From the 18,261 instances, a total of 177,376 HTTP responses and 7,095 HTTP requests were captured. The average amount of uploaded data per `.onion` site was 256B, while the median amount of uploaded data per `.onion` site was 158B. The average amount of downloaded data per `.onion` site was 608KB, while the median amount of downloaded data per `.onion` site was 45KB. The average size of one response was 55KB; the average size of a request was 87B. Clearly the amount of downloaded data surpasses the amount of uploaded data as clients are simply issuing a HTTP request for objects within the server.

The average number of requests to an `.onion` site was 3, while the average number of responses was 11.

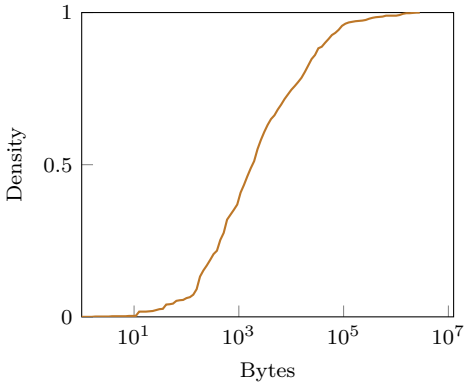

Fig. 4. CDF of the HTTP response size in the 25K crawl (in log scale).

The average size of an `.onion` site then is a little over 608KB. In 2015, the average standard website was just over 2MB, and the average number of objects was over 100 [25, 29], much larger than the average size and number of objects of an `.onion` site. Clearly there is a distinct difference between standard websites and `.onion` sites; standard websites are much larger and contain a greater number of objects within the HTML index page, we note however that the space of all standard websites is orders of magnitude greater than the

7 https://github.com/webfp/tor-browser-crawler
8 https://ahmia.fi

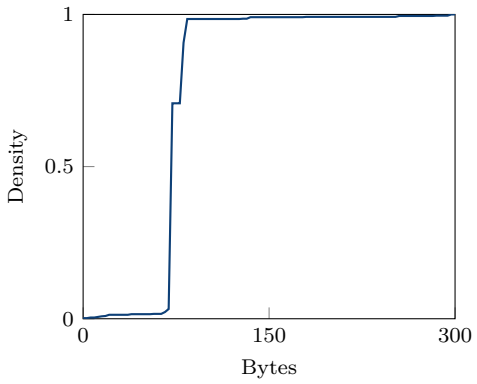

Fig. 5. CDF of the HTTP request size in the 25K crawl.

space of all `.onion` sites and so contains much greater variance in both size and number of objects.

From Figure 5 we see that nearly all HTTP requests were less than 100 bytes, combining this with the knowledge that there are on average just three HTTP requests to download the `.onion` site, we can infer it is most common to download the entire site with just one or two requests after the initial HTTP GET request. From Figure 4, 99% of HTTP responses are less than 1MB in length, and nearly 70% are less than 10KB.

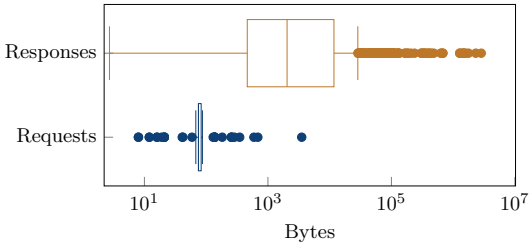

Fig. 6. Boxplot of the HTTP request and response sizes for 25K `.onion` sites.

From Figure 6 we see that the majority of requests are between $70 - 100$B, with relatively few outliers. There is a large skew between the majority of responses of size less than a few KB's and a comparatively (to the number of request outliers) large number of response outliers that are orders of magnitude larger in size than the average response size.

6 Evaluation

To assess the effectiveness of our defenses against WF attacks, we have crawled the same set of pages with and without the defenses in place. Comparing the accuracy of state-of-the-art attacks on both datasets provides an estimate of the protection offered by the defenses.

6.1 P-ALPaCA & D-ALPaCA Evaluation

We evaluate the server-side defenses when a server does not wish to transform its network traffic to look like another `.onion` site but wishes to morph their traffic so it resembles an "average" `.onion` site. We use results from subsection 5.2 to extract information such as the average number of objects and the average size of these objects across all `.onion` sites. A participating server can then use such information to modify their `index.html` page, resulting in an `.onion` site resembling, at the network layer, many different `.onion` sites rather than a specific targeted site.

The object distributions statistics may change over time and require periodic updates. However, to determine whether they change and how often is out of the scope of this paper and leave it for future research. Such an update mechanism could be served by a trusted entity in the Tor network (e.g., a directory authority) that supplies `.onion` sites with this information.

In addition to transforming the network traffic of an `.onion` site to resemble many different "average" `.onion` sites rather than a targeted site, this method allows the server to control the bandwidth overheads at a more fine grained level, since the server can decide the amount and size of extra objects placed in the `index.html` page.

Table 2. P-ALPaCA & D-ALPaCA latency and bandwidth overheads.

	Latency		Volume	
	%	Avg. (s)	%	Avg. (KB)
Undefended	–	3.99	–	175
P-ALPaCA	52.6	6.09	86.2	326
D-ALPaCA (2, 500, 5000)	66.3	6.63	3.66	182
D-ALPaCA (2, 5000, 5000)	56.1	6.22	9.84	193
D-ALPaCA (5, 2500, 5000)	61.7	6.44	15.1	202
D-ALPaCA (10, 5000, 5000)	41.7	5.65	44	254

The server also has control over how often their site is morphed. The frequency of morphing depends on the estimation of how quickly an adversary can mount an attack. If an adversary, can train on network traffic from the server and monitor during a period where the site remains unchanged, the defense will not be of any use. However, the time to train and launch an attack on a number of .onion sites will likely be in the order of hours not minutes[9], as long as a server morphs the site in a shorter period than this, the training data the attacker gathers will be of little use.

To confirm this assertion, we collected 40 network traffic loads, which we call an instance, for each site of 100 .onion sites. We chose 100 .onion sites that resembled the average size of an .onion site[10], in terms of total page size and number of objects. We also collected 40 P-ALPaCA morphed instances for each of the .onion sites, such that each instance is the result of a new morphing process[11] . We then check whether an adversary, training on different morphed versions of an .onion site, can still correctly determine the .onion site of origin.

More specifically, for each of the 100 .onion sites, we collect 40 instances. Resulting in 4000 overall traces. We then apply our server-side defense and re-visit the newly defended sites, resulting in another 4000 traces. We then apply, separately, WF attacks to both undefended and defended .onion sites, training on 60% of traces and testing on the remaining 40%. We consider the defense successful if the WF attack accuracy on the defended .onion sites is dramatically lower than attack accuracy on the undefended .onion sites.

To explore the parameter space, we also evaluated D-ALPaCA, under four different parameter choices. We collected 20 instances for the same 100 .onion sites and compared attack accuracy against both the undefended and P-ALPaCA defended .onion sites. The parameter choices were: λ - the defended page will have a multiple of λ objects, σ - each of the defended page's objects will have a size which is multiple of σ, max_s - when generating new padding objects, sample uniformly within the set [σ, 2*σ, 3*σ, ..., max_s]. Specifically, we chose the following parameter values for (λ, σ, max_s): $(2, 500, 5000)$, $(2, 5000, 5000)$, $(5, 2500, 5000)$, $(10, 5000, 5000)$.

User Experience: in Table 2, we see that average latencies are approximately 40-60% greater in the protected traces than in the unprotected ones. In seconds, the extra time that the user will spend loading the pages is between two and three seconds. We also measured the times to load the original resources in the protected traces with respect to loading all content, since serving extra padding resources once all the original content is sent does not impact on user experience. We call the time between the first request to the last legitimate request *UX-time*. However, the average difference between UX-time and the time to load all resources in a protected page is less than 200ms. We notice that the randomization of RP often sends original requests at the end of the transmission which explains the mild difference between UX-time and total page load time.

Table 3. Closed world classification for .onion sites morphed via P-ALPaCA and D-ALPaCA, with other defenses added for comparison. CUMUL depends on packet lengths and so some defenses that only operate on packet time information cannot be applied.

	k-NN (%)	k-FP (%)	CUMUL (%)
Undefended	45.6	69.6	55.6
P-ALPaCA	0.2	9.5	15.6
D-ALPaCA (2, 500, 5000)	9.5	22.7	27.0
D-ALPaCA (2, 5000, 5000)	12.5	34.4	40.0
D-ALPaCA (5, 2500, 5000)	5.8	22.3	30
D-ALPaCA (10, 5000, 5000)	7.2	22.9	33.0
Decoy [21]	4.9	11.2	X
Tamaraw [4]	6.8	14.0	X
BuFLO [9]	5.3	13.3	X

Closed World classification: we performed a closed world WF attack on P-ALPaCA defended, D-ALPaCA defended and undefended .onion sites. If our server-side defenses are successful, defended .onion sites should, at the network level, look similar to one another and result in a low classification accuracy. We

9 For example, we used a total of 100 .onion sites in experiments, visiting each .onion sites 40 times. We trained on 60% of data. The average page load time was around 4 seconds. Therefore an attacker, using one machine for crawling and gathering training data, would be able to initiate an attack after 9600 seconds. However, we note an attacker can parallelize this process for faster attacks.

10 Via section 5.

11 As proposed in subsection 4.1, the .onion site is differently morphed upon every client visit.

Table 4. Open world classification for .onion sites morphed P-ALPaCA and D-ALPaCA.

	k-NN (%)		k-FP (%)		CUMUL-k-FP (%)	
	TPR	FPR	TPR	FPR	TPR	FPR
Undefended	37.0	1.0	62.1	0.8	49.7	5.4
P-ALPaCA	0.4	0.2	3.6	0.2	1.1	1.3
D-ALPaCA (2, 500, 5000)	4.5	0.2	12.0	0.4	21.4	1.4
D-ALPaCA (2, 5000, 5000)	7.5	0.4	12.6	0.4	28.8	1.2
D-ALPaCA (5, 2500, 5000)	6.0	0.3	12.7	0.3	18.7	1.3
D-ALPaCA (10, 5000, 5000)	3.4	0.3	13.3	0.3	27.3	1.0

use CUMUL [20], *k*-FP [11] k-NN [30] for evaluation[12]. The number of neighbours used for classification is fixed at two.

Table 3 shows the closed-world classification results of undefended .onion sites against .onion sites with each instance uniquely defended using P-ALPaCA or D-ALPaCA. WF attacks are ineffective under both defenses, and in fact P-ALPaCA improves upon *Tamaraw* and *BuFLO*. D-ALPaCA does slightly worse than the P-ALPaCA in terms of defending .onion sites, but as can be seen from Table 2, has real advantages in terms of limiting bandwidth overheads. For example, D-ALPaCA with parameters (2, 500, 5000), reduced k-FP accuracy from 69.6% to 22.7%, compared to the P-ALPaCA which reduced attack accuracy to 10%. But, D-ALPaCA (2, 500, 5000) required 23.6 times less bandwidth than P-ALPaCA to achieve these results. A server operator wishing to provide a defense to its clients while limiting the increase in bandwidth may then consider this a worthwhile trade-off and choose to use D-ALPaCA over P-ALPaCA.

Open World classification: in addition to closed world experiments, we evaluated the server-side defenses in the open world setting, where we include network traffic instances of .onion sites that are not of interest to the attacker. We observe how the classification accuracy is affected in this setting, which is intended to reflect a more realistic attack. We use 5,259 unique .onion sites, from subsection 5.2, as background traffic instances[13] and set the number of neighbours used for classification at two. Note that CUMUL only does binary classification in the open world, classifying as either a background instance or a foreground instance of interest, whereas *k*-FP and k-NN attempt to classify an instance to the correct .onion site if it is flagged as a non-background instance. In order to compare the results of the attacks in the open-world, we have used the feature vectors of CUMUL while applying the k-FP classification process. To make sure that the classification model does not affect the accuracy of the attack, we evaluated the CUMUL features with k-FP in a closed-world and achieved a similar accuracy to SVM.

As we can see from Table 4 there is a dramatic decrease in attack accuracy when both P-ALPaCA and D-ALPaCA are used, showing that if a server morphs their site at a higher rate than the adversary can gather training data, the site will be almost perfectly concealed.

D-ALPaCA parameter choices: Table 3 and Table 4 show there is no notable difference in attack accuracy when changing parameters. However, as expected, smaller parameter choices led to smaller bandwidth overheads.

6.2 LLaMA Evaluation

We have crawled the same list of .onion sites as in the evaluation of ALPaCA, under four different conditions:

JS enabled: we collected our data with no defense installed and JavaScript enabled, the default setting in the Tor Browser.

JS disabled: we repeated the same crawl as with JS enabled but disabling JavaScript in the Tor Browser. We keep JS disabled for the rest of our crawls.

RP with delays: we collected data with the defense only delaying requests, altering the order of the requests as described in section 4.

Extra requests: we crawled the same data with the defense adding delays and extra requests as described in the previous section.

We note that we have disabled RP in the Tor Browser for all the crawls above by disabling the browser preference network.http.pipelining.

In Table 5, we show the results for the three classifiers in the closed world of 100 onion sites. We do not observe much difference in accuracy between JavaScript enabled and disabled. This shows that our assumption

12 We use Tobias Pulls' implementation of the k-NN website fingerprinting attack [26].

13 For *k*-FP, we train on 1,000 of the 5,259 background traces and for each .onion site we train on 50-75% of instances. Whereas k-NN uses Leave-one-out cross-validation on the entire dataset.

of no dynamic content holds for the list of onion sites used in our evaluation.

Table 5. Closed world classification for .onion sites under different countermeasures.

	k-NN (%)	k-FP (%)	CUMUL (%)
JS enabled	64.0	55.8	52.4
JS disabled	60.8	53.4	52.7
RP with delays	46.8	47.9	49.6
Extra requests	31.5	36.0	34.8

When the defense only adds delays to requests, the accuracy of the classifiers decreases 10% in the k-NN classifier and has limited effect on k-FP and CUMUL. The mild impact on the accuracy of the classifier may imply that the hypothesis that RP does not work because servers do not support HTTP pipelining does not hold, suggesting that the request randomization strategy is flawed, as previous evaluations have argued [5, 31].

We also evaluated the scenario in which the countermeasure, besides adding delays, repeats previous HTTP requests. We observe a significant decrease in accuracy to almost half the accuracy obtained in the unprotected case for the k-NN classifier.

In Table 6, we show the overheads of LLaMA for its two different modes. We see that overheads are around 10%. Even though the protection provided by the defense is considerably lower than the server-side defense or other defenses in the literature, its simplicity and the small overhead that it introduces makes it a good candidate for a WF countermeasure.

Table 6. Latency and bandwidth overheads of the client-side defense in the closed world.

	Latency		Volume	
	%	Avg. (s)	%	Avg. (KB)
JS disabled	–	5.01	–	126
RP with delays	8.4	5.42	X	X
Extra requests	9.8	5.49	7.14	135

7 Discussion and Future Work

Both the ALPaCA and LLaMA have performed at least as well as state-of-the-art defenses, showing that application layer WF defenses do indeed protect against attacks. Next we discuss potential avenues for future research.

Ease of Deployment. We argue that application layer defenses are simpler to implement than previously proposed approaches as they require no modifications to existing protocols or participation from a relay in the circuit. The only expensive part of ALPaCA comes in the form of the gathering of statistics for the probabilistic based morphing approach. However, we suggest this cost can be amortized over all participating servers by allowing a centralized entity to collect this information, such as is done by directory authorities now to collect Tor relay descriptors. Future research could determine how often these statistics must be updated. Implementation of the client-side defense is simple, as we developed it as a browser add-on. This could be made available to Tor clients either by direct integration in to the Tor browser bundle, or through an add-on store.

Rate of Adoption. Initially, we expect relatively few .onion sites to implement server-side defenses. Over time if a significant number of .onion sites adopt ALPaCA, it is possible that a large fraction of sites will morph their page to resemble one another. In turn, this will create stable anonymity sets of .onion sites that have the same network traffic patterns. Finding the rate and size of these anonymity sets is left for future work.

Clearly, smaller .onion sites are easier to protect than larger ones, as it is impossible to morph a larger site to resemble network traffic patterns of a smaller site. Thus, we expect larger .onion sites to be more difficult to protect over time. However, as subsection 5.2 show, the majority of .onion sites are small and so should be relatively simple to defend against WF attacks.

Latency and Bandwidth Overheads. All WF defenses come at the expense of added latency and bandwidth. Our defenses allow the exact overheads to be tuned by the participating client or server. We saw from subsection 6.1 that P-ALPaCA adds, on average, 52.6% extra waiting time and 86.2% additional bandwidth. We note, that compared to previous works, these overheads are relatively small, and that due to the nature of .onion sites, even the morphed pages are small in size compared to standard web pages. LLaMA improves

on striking a balance between overhead limitation and protection against WF attacks. By issuing additional HTTP requests, WF attack accuracy is halved, while only adding 9.8% in waiting time and 7.14% in bandwidth. We also saw comparably small overheads in our D-ALPaCA defense which significantly reduced WF attack accuracy at the expense of an additional 3.66% of bandwidth.

Natural WF Defenses. We note that compared to related works, the attack accuracy on `.onion` sites seems alarmingly low. Wang et al. [30] achieved accuracies of over 90% when fingerprinting the top 100 Alexa websites, whereas our experiments on 100 `.onion` sites resulted in an accuracy of only 45.6% using the same classifier. We have validated the results of Wang et al. on the top 100 Alexa websites, removing the possibility of a bug or some irregularity in our own crawler. We conclude that this reduction in accuracy is an artifact of the size and type of the majority of `.onion` sites. The average size of a `.onion` site is substantially smaller than that of a standard web page; resulting in less information being leaked to a classification process, allowing for the increase in chance of misclassifications. We also found that a large number of `.onion` sites are log-in pages to various forums, that are based on standard designs and so bear a resemblance to one another. The small size and design of `.onion` sites provide a natural defense against WF. By restricting the amount of information available to a classification process, and conforming to standard website designs, despite the small world size of `.onion` sites we conclude that successful website fingerprinting attacks are considerably more difficult than on standard websites.

HTTP/2. HTTP/2 is the upcoming new version of the HTTP protocol and is already supported by some of the domains that receive most traffic volume in the Web [1]. HTTP/1.1 tried to provide parallelism of HTTP messages with HTTP pipelining. However, the deployment of HTTP pipelining has not been ideal, as many intermediaries (e.g., CDNs) do not implement it correctly [1]. HTTP/2 supports parallelism of HTTP conversations natively and overcomes one of the main limitations of HTTP/1.1. From our experiments with request randomization performed with LLaMA, our intuition is that randomization of HTTP/2 will not provide better results than RP. HTTP/2 also allows to add padding in HTTP messages to mitigate cryptographic attacks [10]. We devise the use of HTTP/2 padding as a primitive for application-layer WF defenses.

8 Conclusion

We proposed two WF defenses for `.onion` sites, a server-side defense and a client-side defense, that operate at the application layer. The choice of working at this layer has the following benefits: i) it gives fine control over the content of webpages, which is arguably the reason why WF attacks are possible, and ii) it makes the defenses easy to implement.

The server-side defenses morph the content of a webpage before it is loaded by a client. The resulting webpage looks exactly as the original in terms of its visual content, but it behaves as a completely different page at the network level, where the WF adversary sits. Intuitively, since the adversary will observe a different webpage for each load, they will not be able to perform the attack. Experiments on `.onion` sites confirm this intuition, and show that this defense effectively reduces the accuracy of an adversary.

We have designed and evaluated a lightweight client-side defense at the application layer. The evaluation shows that this defense reduces the accuracy of the attack in the onion world significantly and, even though it offers lower protection than the server-side defenses, it provides a high security versus overhead ratio. Furthermore, its simplicity and its implementation as an add-on for the Tor Browser favor its deployment in the live Tor network.

Acknowledgments

We thank the anonymous reviewers for their useful comments. A special acknowledgment to Gunes Acar, George Danezis, Mustafa A. Mustafa and Noah Veseley, for their feedback to improve this research. This material is based upon work supported by the European Commission through KU Leuven BOF OT/13/070, H2020-DS-2014-653497 PANORAMIX and H2020-ICT-2014-644371 WITDOM and partly supported by the UK Government Communications Headquarters (GCHQ), as part of University College London's status as a recognised Academic Centre of Excellence in Cyber Security Research. Giovanni Cherubin was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1). Marc Juarez is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO).

References

[1] HTTP/2 specs. "https://http2.github.io/", 2015. (accessed: August, 2016).

[2] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Workshop on Privacy in the Electronic Society (WPES)*, pages 121–130. ACM, 2014.

[3] X. Cai, R. Nithyanand, and R. Johnson. Glove: A Bespoke Website Fingerprinting Defense. In *Workshop on Privacy in the Electronic Society (WPES)*, pages 131–134. ACM, 2014.

[4] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *ACM Conference on Computer and Communications Security (CCS)*, pages 227–238. ACM, 2014.

[5] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *ACM Conference on Computer and Communications Security (CCS)*, pages 605–616. ACM, 2012.

[6] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy (S&P)*, pages 191–206. IEEE, 2010.

[7] H. Cheng and R. Avnur. Traffic Analysis of SSL Encrypted Web Browsing. *Project paper, University of Berkeley*, 1998. Available at http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps.

[8] R. Dingledine, N. Mathewson, and P. F. Syverson. "Tor: The Second-Generation Onion Router". In *USENIX Security Symposium*, pages 303–320. USENIX Association, 2004.

[9] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *IEEE Symposium on Security and Privacy (S&P)*, pages 332–346. IEEE, 2012.

[10] Y. Gluck, N. Harris, and A. Prado. Breach: reviving the crime attack. *Unpublished manuscript*, 2013.

[11] J. Hayes and G. Danezis. k-fingerprinting: a Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium*. USENIX Association, 2016.

[12] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *ACM Workshop on Cloud Computing Security*, pages 31–42. ACM, 2009.

[13] A. Hintz. Fingerprinting Websites Using Traffic Analysis. In *Privacy Enhancing Technologies (PETs)*, pages 171–178. Springer, 2003.

[14] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A critical evaluation of website fingerprinting attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 263–274. ACM, 2014.

[15] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. Toward an Efficient Website Fingerprinting Defense. In *European Symposium on Research in Computer Security (ESORICS)*, pages 27–46. Springer, 2016.

[16] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: passive deanonymization of tor hidden services. In *USENIX Security Symposium*, pages 287–302. USENIX Association, 2015.

[17] M. Liberatore and B. N. Levine. "Inferring the source of encrypted HTTP connections". In *ACM Conference on Computer and Communications Security (CCS)*, pages 255–263. ACM, 2006.

[18] L. Lu, E. Chang, and M. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In *European Symposium on Research in Computer Security (ESORICS)*, pages 199–214. Springer, 2010.

[19] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Network & Distributed System Security Symposium (NDSS)*. IEEE Computer Society, 2011.

[20] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website fingerprinting at internet scale. In *Network & Distributed System Security Symposium (NDSS)*. IEEE Computer Society, 2016.

[21] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114. ACM, 2011.

[22] M. Perry. Committed to the official Tor Browser git repository, https://gitweb.torproject.org/tor-browser.git/commit/?id=354b3b.

[23] M. Perry. Experimental Defense for Website Traffic Fingerprinting. Tor project Blog. "https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting", 2011. (accessed: October 10, 2013).

[24] M. Perry, G. Acar, and M. Juarez. personal communication.

[25] A. Pinto. Web Page Sizes: A (Not So) Brief History of Page Size through 2015. yottaa.com. "http://www.yottaa.com/company/blog/application-optimization/a-brief-history-of-web-page-size/", 2015. (accessed: April 18, 2016).

[26] T. Pulls. A golang implementation of the kNN website fingerprinting attack. "https://github.com/pylls/go-knn", 2016. (accessed: May, 2016).

[27] SecureDrop. securedrop.org. "https://securedrop.org/", 2016. (accessed: April 20, 2016).

[28] Q. Sun, D. R. Simon, and Y. M. Wang. Statistical Identification of Encrypted Web Browsing Traffic. In *IEEE Symposium on Security and Privacy (S&P)*, pages 19–30. IEEE, 2002.

[29] MobiForge. mobiforge.com. "https://mobiforge.com/research-analysis/the-web-is-doom", 2016. (accessed: April 20, 2016).

[30] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*, pages 143–157. USENIX Association, 2014.

[31] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 201–212. ACM, 2013.

[32] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Network & Distributed System Security Symposium (NDSS)*. IEEE Computer Society, 2009.

A Onion service target experiments

In addition to morphing a page via P-ALPaCA and D-ALPaCA, we evaluate the efficacy of our server-side defense on a number of .onion sites via morphing to a target .onion site. We dispense with applying a new morphing process for each capture of an .onion site load. Instead, we morph the .onion site once and capture 40 instances of this morphed site. We show that even if a server morphs their network traffic once, if it is morphed towards a targeted .onion site, this is enough to thwart WF attacks.

A.1 SecureDrop

To protect its users, SecureDrop may want to morph the network traffic pattern of its page load to look like that of an .onion site which would not raise suspicion on a visit. We collected 40 instances of network traffic when visiting SecureDrop; we then chose 40 target .onion sites which our server-side defense would morph SecureDrop's traffic to look like.

We considered a powerful adversary, who knows all sites that the defense would like to morph traffic to look like. For each target site, the adversary could train on all the undefended SecureDrop network traffic and the network traffic of the target .onion site, and they must classify an unknown traffic instance as either SecureDrop or the target .onion site. In our experiment, all new traffic instances were the morphed SecureDrop page; under a perfect defense all should have been classified as the target site

Using k-FP with 1,000 trees [11], the average binary classification accuracy over the 40 different .onion sites was 0.372 ± 0.416. Overall, our server-side defense was successful in obscuring which site a client was visiting, though we saw a large variation: some onion sites perfectly concealed the true label while others failed.

The average communication cost (incoming and outgoing size of packets) of the SecureDrop page was 15 KB, and it loaded on average in 4.62 seconds. The average communication cost of the morphed page was 373 KB and it loaded in 6.70 seconds. The size of the morphed page entirely depends on the target page we chose to morph the SecureDrop page towards, if a smaller target page had been chosen this would result in a smaller bandwidth overhead. However, the average bandwidth overhead is still smaller than that of a standard website.

A.2 Facebook

To generalize our defense beyond SecureDrop we chose 100 .onion sites that may also wish to protect visiting clients from WF attacks, by morphing their traffic to that of the Facebook .onion site[14]. We collected 40 traffic instances for each .onion site. All WF attacks were applied in the same manner as in subsection 6.1.

Binary classification: the average binary classification accuracy over the 100 .onion sites was 0.098 ± 0.253. Even when the adversary knows undefended and target site, the attack's accuracy is below 10%.

Closed World classification: we also compared a closed world attack on the 100 undefended .onion sites and the same attack after morphing those sites to look like Facebook .onion site. If our server side defense is successful the 100 morphed .onion sites should, at the network level, look like the Facebook .onion site, resulting in a low classification accuracy.

Table 8, shows as expected, attack accuracy decreases when onion sites are morphed to resemble Facebook's network traffic patterns.

Table 7. Facebook experiment latency and bandwidth overheads.

	Latency		Volume	
	%	Avg. (s)	%	Avg. (KB)
Undefended	–	3.99	–	175
Defended	27.3	5.08	80	315

Table 8. Closed world classification for .onion sites morphed to Facebook's .onion site.

	k-NN (%)	k-FP (%)	CUMUL (%)
Undefended	45.6	69.6	55.6
Defended	9.4	55.6	53.6

Open World classification: in addition to closed world experiments, we evaluated the server-side defense in the open world setting, where we included instances

14 https://facebookcorewwwi.onion

of .onion sites that were not of interest to the attacker. We used 5,259 unique .onion sites, from subsection 5.2, as background traffic instances. Table 9 shows, as expected, attack's accuracy decreases when sites are morphed to resemble Facebook's network traffic patterns.

Table 9. Open world classification for .onion sites morphed to Facebook's .onion site.

	k-NN (%)		k-FP (%)		CUMUL-k-FP (%)	
	TPR	FPR	TPR	FPR	TPR	FPR
Undefended	30.8	2.6	59.3	5.2	53.2	5.7
Defended	7.8	0.9	44.9	1.8	44.4	2.0

Table 7 shows the average time to load a page only increases by 1.09s when morphing a page to the Facebook .onion site. We also see that the bandwidth overhead is, compared to previous works, quite tolerable. The total cost of communication rises by only 140KB.

B KDE distributions

We used Kernel Density Estimation (KDE) to estimate the distributions of number of objects (Figure 7), size of html pages (Figure 8) and size of objects (Figure 9). KDE is a non-parametric method for estimating a probability distribution given a data sample, which provides smoother estimates than histograms. KDE requires to specify a kernel (Gaussian, in our case) and a bandwidth. The bandwidth impacts on the smoothness of the estimate: a larger bandwidth tends to provide better smoothness, but less fidelity to the original data. To determine the bandwidth for each of our distributions, we first performed Grid Search Cross Validation using scikit-learn library[15], to obtain a rough idea of the bandwidth ranges. Then, we manually trimmed the bandwidth to achieve what visually seemed to reflect well the variance of data, but also provided smooth distributions. For our purposes, it was important to have smooth estimates to guarantee a good quality in sampling (e.g., to avoid spikes). We used a bandwidth of 2 for the distribution over objects, and of 2000 for both the HTML and object sizes distributions.

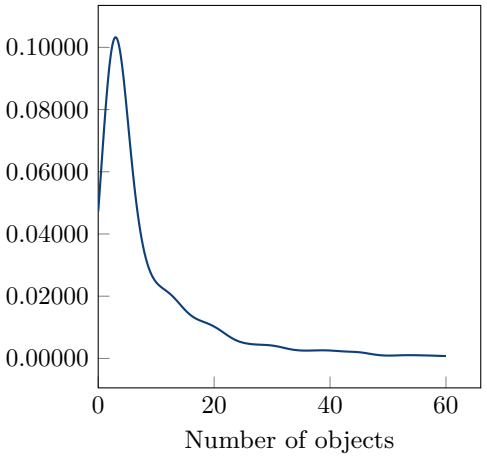

Fig. 7. KDE distribution of the number of objects

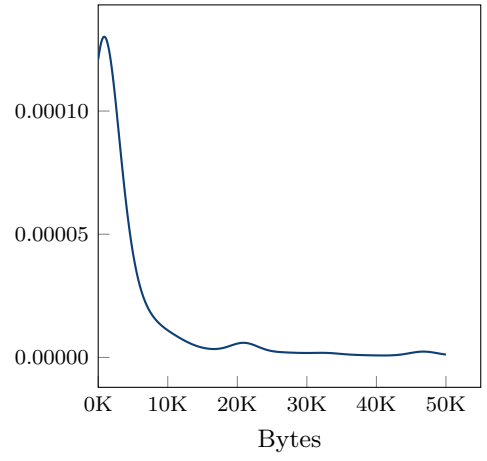

Fig. 8. KDE distribution of the HTML sizes

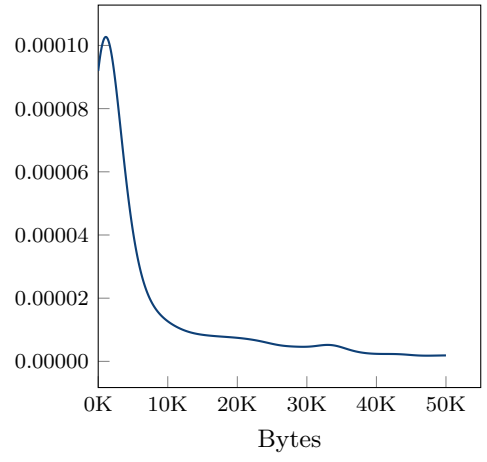

Fig. 9. KDE distribution of the object sizes

15 http://scikit-learn.org/