Evaluating the Efficacy of Gaussian Padding on Website Fingerprinting Attacks

Master thesis by Johannes Leupold
Date of submission: 01.09.2021

1. Review: Jean-Paul Degabriele
2. Review: Vukasin Karadzic

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

IT Security

Cryptography and Network Security

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Johannes Leupold, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 01.09.2021

J. Leupold

Abstract

Abstract

Contents

1 Introduction

This is the introduction.

2 Background Material

2.1 Theoretical Setting

Website Fingerprinting (WF) is a field of study focusing on data leaked from anonymization techniques and whether it is still sufficient to conclude which websites the user visited. Furthermore, a goal of Website Fingerprinting study is the evaluation of various countermeasures that limit the leaked data and thus thwart attempts to uncover the websites a certain user was visiting.

2.1.1 Threat Model

The Website Fingerprinting adversary is assumed to have access to the encrypted network traffic of the victim. However, he is not able to decrypt the traffic, or parts of it, so he can only draw conclusions based on metadata of the encrypted packets. In particular, the adversary observes

- packet length (full packet size in bytes),
- direction (whether the packet goes from Client $\rightarrow$ Server, or vice versa), and
- timing (amount of time between the first packet and the current packet).

This information also gives way to various derived measures, such as overall bandwidth consumed, overall time, or burst[1] length/size.

Before the adversary begins with the attack, he can create a database of websites the victim may visit and collect so-called *packet traces* for the websites. A packet trace is a sequence $\mathcal{T} = \langle p_1, p_2, ..., p_n \rangle$ of tuples $p_i = (t_i, l_i, d_i) \in \mathbb{N} \times \{52, \ldots, MTU\} \times \{\uparrow, \downarrow\}, i = 1, \ldots, n,$

[1]A *burst* is a contiguous sequence of packets going in the same direction

where t_i, l_i and d_i denote the time, length and direction for a single observed packet, respectively. The packet length can never be smaller than 52 bytes, as that is the size of the smallest possible TCP/IP packet, the ACK packet. The maximum packet size MTU (also called *Maximum Transmission Unit*) is dependent on the used network technologies. For Ethernet, the MTU is 1500 bytes, while for WiFi (IEEE 802.11) it is 2312 bytes. Note that the adversary is assumed to collect his database of website traces under the same technical preconditions (network technologies) that his victim uses to access them. When carrying out the attack, the adversary observes packet traces for particular page loads of the victim. He may then use any algorithm and his prepared database to try and deduce which websites out of the pre-selected set the victim visited. When a WF countermeasure is in place, the adversary is assumed to be aware of it such that he can collect his trace database under the same countermeasure.

WF threats may be considered either in the Closed World or the Open world scenario. In the Closed World, the goal of the adversary is to identify the websites a victim visited from a fixed set of websites known in advance, while in the Open World scenario, the adversary defines a set of monitored web pages and intends to identify whether the victim visited a website from that set.

2.1.2 Attacks

To successfully make conclusions on the websites a victim visited, the attacker needs to classify the packet traces he observed to be an instance of one of the websites in his database. He may achieve this by randomly guessing the instance's class, but this naive method can't be expected to yield good results for higher numbers of websites. Therefore, the adversary may employ a Machine Learning technique, consisting of a feature extraction[2] and a classification algorithm. A classification algorithm is trained using a number of examples from all classes including the class labels (the *training set*). In the following, an overview of common classification algorithms is given.

Nearest Neighbors

The Nearest Neighbors classifier (also called NN) is a simple algorithm that assigns to each instance the class of its nearest neighbor according to some distance metric. It may

[2]*Feature Extraction* refers to transforming a certain observation into a number of (mostly numerical) attributes, called *features*

be extended to k-Nearest-Neighbors (k-NN) by considering the nearest k neighbors and taking the majority vote of their classes. One may also apply weighting to the neighbors based on the distance from the point to be classified. The NN classifier is an instance of k-NN for $k = 1$.

Training: The NN classifier is trained using a number of data points and their respective classes, which it stores internally. Also, a distance metric is selected.

Classification: Being presented a data point, the k-NN classifier looks for the closest k neighbors of the point among its stored training points with respect to the chosen distance metric. It then returns the class label which the majority of those neighbors have.

Naive Bayes

The Naive Bayes classifier tries to estimate probability distributions for the attribute values conditioned on all classes from the training data. To classify a new example, Bayes' rule is used to calculate the conditional probability of the example for each class C and the class yielding the highest probability is returned.

Training: The NB classifier is trained using a number of data points and their respective classes. It may estimate the prior probability of the classes $p(C)$ from their relative frequency in the data or use a predefined prior (e.g. uniform). Then it estimates a conditional probability distribution on data attributes $p(x \mid C)$ for every class C. This may be done by fitting a normal distribution to the data using a maximum likelihood estimate (or any other probability distribution that is considered to fit the data), or by employing Kernel Density Estimation (see below).

Classification: To classify a new data point x, the NB classifier uses Bayes' rule to calculate the conditional probability of each class given the point:

$$p(C \mid x) = \frac{p(x \mid C) \cdot p(C)}{p(x)}$$

The probabilities $p(C)$ and $p(x \mid C)$ are known due to the training. $p(x)$, the total probability of point x, is the same for all classes, hence the following proportionality holds

$$p(C \mid x) \propto p(x \mid C) \cdot p(C)$$

After calculating $p(x \mid C) \cdot p(C)$ for all classes C, the NB classifier returns the class that yields the highest such value.

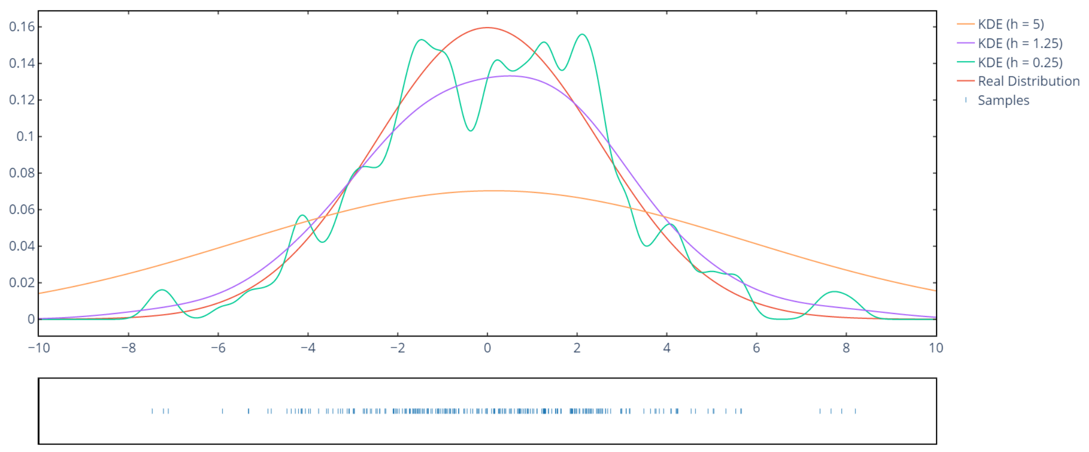

Figure 2.1: Result of applying KDE with different bandwidths, real distribution and samples are shown

Kernel Density Estimation (KDE) The term ”Kernel Density Estimation” refers to a method used to estimate the underlying probability density function a certain set of samples was drawn from. As opposed to maximum likelihood methods that require the distribution's type to be known in advance, KDE is able to approximate any probability distribution. The accuracy of this approximation depends in large parts on the smoothing parameter h, called the *bandwidth*, which controls the area of influence a single data point has on the resulting probability density. See Figure 2.1 for an example. When evaluating the estimated density function at a certain point, a so-called *kernel* function is centered over each training sample, and evaluated to obtain the probability density. A kernel is a non-negative function, which is usually chosen to be symmetric and centered on zero[3]. A common choice is the standard normal distribution, although there are many other options as well[4].

Training: The Kernel Density Estimator is trained with a number of samples that are stored internally. Additionally, a kernel is selected.

[3]Most of the time, the kernel should itself be a probability density, leading the estimated density to be a valid probability density as well

[4]See `https://scikit-learn.org/stable/modules/density.html`, Section 2.8.2 for a non-exhaustive list

Estimation: To estimate the density at a certain point x, the individual centered kernels for every training example x_i are evaluated at the point x. Subsequently, the results are summed and scaled to get valid probability values. Equation 2.1 shows the full formula.

$$p(x) = \frac{1}{nh} \sum_{i=1}^{n} k\left(\frac{x - x_i}{h}\right) \tag{2.1}$$

Support Vector Machine (SVM)

The SVM is a sophisticated classification algorithm trying learn a hyperplane to separate the examples of different classes in the feature space. This is achieved by selecting the hyperplane that yields the largest separation between the classes, i.e. the one that maximizes the distance (the *margin*) to the nearest points of each class. Intuitively, this leads to a better generalization ability to unseen examples, as class separation is maximal. Support Vector Machines are well suited for working with high-dimensional feature spaces and may use a non-linear transformation of data points into an implicit higher-dimensional (and even infinite-dimensional) feature space. This transformation is called a *kernel* (not to be confused with the kernel used in Kernel Density Estimation). An advantage of SVMs is the possibility of sparse representation, as only training examples on, within, or on the wrong side of the margin, called *support vectors*, need to be retained for constructing the hyperplane. A cost parameter C penalizes support vectors that are inside the margin or on the wrong side of the hyperplane. See Figure 2.2 for an example. [1, 6].

Training: During training, the SVM solves the optimization problem of finding the hyperplane that best separates the data while maximizing the margin. This is done with respect to the penalty parameter C. The training examples on or within the margin and on the wrong side of the hyperplane are designated as support vectors. All other training examples may be deleted.

Classification: The separating hyperplane is fully defined given the support vectors. After reconstructing the hyperplane, the example is assigned a class according to on which side of the hyperplane it lies.

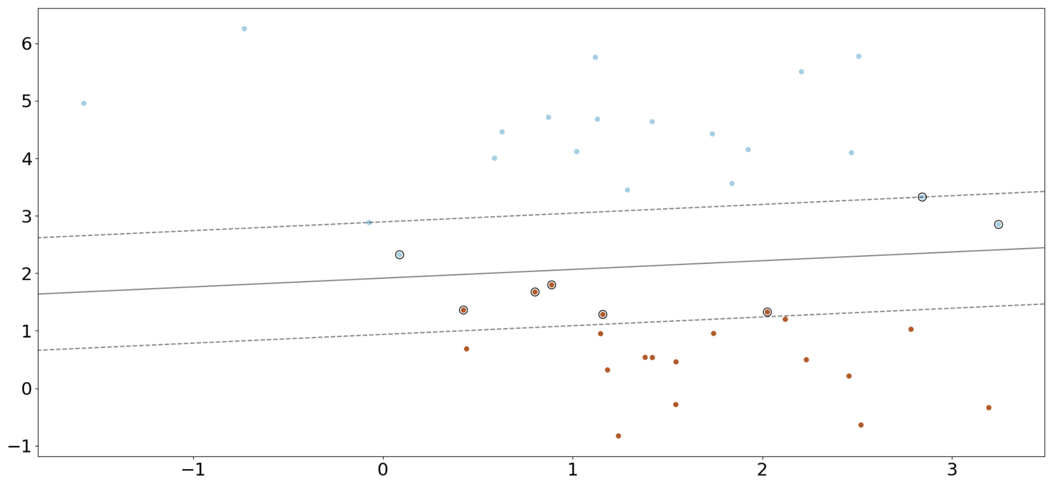

Figure 2.2: The decision surface of a SVM (with $C = 1$), shown in solid gray and the margin (shown in dashed gray). Training examples are colored by their class, while support vectors are outlined by a black circle (taken from [23] with modifications).

2.1.3 Countermeasures

A defense against WF attacks aims at disguising the identifying patterns in the packet traces the attacker can observe. To this end, packets lengths may be artificially changed, they may be split up into multiple packets, or the timing might be changed. There exist three main classes of defenses, while, naturally, some countermeasures might not fit one particular class, but exhibit characteristics of multiple classes.

Padding Padding schemes are among the simplest countermeasures. They add certain amounts of dummy data to each packet up to the MTU. Padding schemes may be *deterministic* or *probabilistic*, with deterministic schemes always returning the same output trace when presented the same input trace, while this isn't the case for probabilistic padding. When using this type of defense, packets can only grow, but never shrink. Also, packet counts and timing are not affected.

Noise Defenses of this class try to disguise the visited web page by deliberately adding noise to the transmission. This can be done for example by loading a second randomly chosen web site in parallel to the requested site [19] or randomizing the order of HTTP requests in the web browser [21]. Countermeasures using approach may affect packet timing and order and insert new packets into the trace.

Morphing Morphing countermeasures are the most complex defenses, trying to hide the actual distribution of packet sizes. They may simply try to make all websites look equal, or at least similar, (cf. [9, 4, 3]) or to make a certain web site look like a different one [28]. Packet traces may be changed in many ways when using a morphing defense, like adding extra packets, growing or splitting packets, or changing timing.

2.2 Gaussian Padding

Gaussian padding is a specific form of probabilistic padding. The amount of dummy data added to a packet is drawn from a rounded normal distribution, as opposed to Uniform Padding, where padding sizes are drawn according to the uniform distribution. Furthermore, the tails of the distribution are clipped on both sides, as padding can never take negative values and packets may never become bigger than the MTU. The distribution is parameterized with the desired mean padding size, becoming the mean μ of the normal distribution. The standard deviation σ can be chosen such that the truncated tail corresponds to as little probability mass as desired. For our purpose, choosing $\sigma = \frac{\mu}{3}$ is sufficient, as in this case only approximately 0.1% of the probability mass is allocated to negative values. Figure 2.3 shows an example distribution function for a truncated rounded normal distribution.

Sampling from a rounded normal distribution is straight-forward by simply sampling from a continuous normal distribution and rounding the value. The truncation can be achieved by rejecting and resampling values that fall out of the desired interval, or by using the method described by Hülsing et al. in [14], which allows to sample from a rounded Gaussian in constant time, thus mitigating cache timing attacks on the sampling algorithm. While these are out of scope under the threat model of WF, this fact is of particular interest for different fields of research, such as Lattice-based cryptography. [14]

When applying Gaussian padding, the amount of data added to each packet may either be sampled from the distribution independently for each individual packet or once for every session (i.e. page load), leading to *packet-random* Gaussian padding and *session-random* Gaussian padding, respectively.

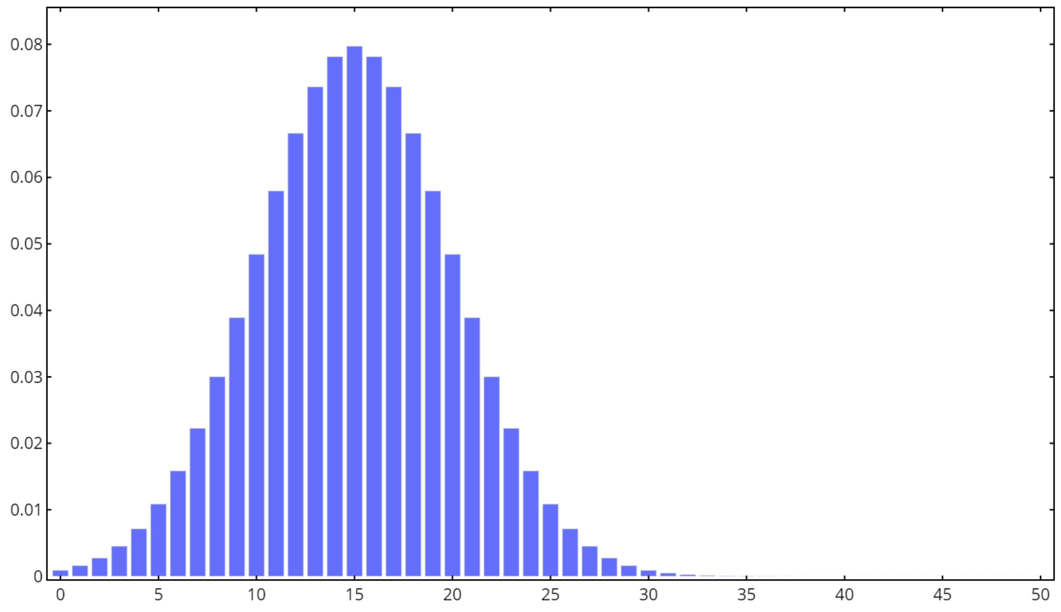

Figure 2.3: An example for a rounded normal distribution, truncated to the left at 0 with a mean padding size of $\mu = 15$ and a standard deviation of $\sigma = 5$

3 Prior Work

Website fingerprinting attacks and defenses have been subject to an ”arms race which has continued for more than a decade” [5]. In the following sections, a selection of major WF attacks (Section 3.1) and defenses (Section 3.2) are presented. Furthermore, Section 3.3 gives an overview over a method proposed by Cherubin [5] to estimate provable security bounds for WF defenses with respect to a given feature set.

3.1 Website Fingerprinting Attacks

Many WF attacks have been published during the last 15 years, with each one improving on the accuracy and performance (including robustness against countermeasures) of the previous attacks. A selection of notable such attacks is presented below, in order of increasing recency. While the early publications still focused on fingerprinting encrypted traffic on SSH tunnels (cf. [15, 13, 19, 9]), with the rise in popularity of the Tor anonymization platform[5] the focus in the WF community has since shifted heavily towards Tor traffic in the recent years (cf. [9, 27, 18, 12, 24, 26]). Tor traces pose additional challenges to the attacker, as in Tor all packets have equal length and some countermeasures have already been deployed [21]. Starting from the *H* attack below, all subsequent attacks are increasingly focused on Tor, with everything beyond *VNG++* no longer considering SSH traffic at all.

LL In [15], Liberatore and Levine present two distinct attacks, one based on a modified Jaccard coefficient (a measure for set similarity), and a second one using a Naive Bayes classifier with KDE. While both attacks have been found to yield good results, the Jaccard classifier's accuracy degrades more rapidly in the face of countermeasures [15]. Because

[5]`https://www.torproject.org/`

of this, most of subsequent work only considered the Naive Bayes classifier, which will be termed *LL* going forward. The attack of Liberatore and Levine uses as features a histogram of packet sizes and directions, i.e. counting the numbers of packets having each size going in each direction. Apart from their contributions to WF attacks, Liberatore and Levine have also published a dataset of website traces for WF evaluation (see Section 4.1 for more details).

H The attack by Herrmann et al. [13] uses the same feature set as the LL attack, but instead of using the Naive Bayes classifier, it applies a set of well-known text mining transformations on the packet size histogram, namely the *TF (term frequency) transformation* (equation 3.1) and *cosine normalization* (equation 3.2). Then, they apply the Multinomial Naive Bayes classifier, which is commonly used in text mining [13]. A detailed description of the classifier can be found in [16, ch. 13].

$$f^*_{x_i} = log(1 + f_{x_i}) \tag{3.1}$$

The x_i values denote pairs of packet directions and sizes $(d, l) \in \{\uparrow, \downarrow\} \times \{52, \ldots, MTU\}$, while f_{x_i} is the absolute frequency of such packets in the trace. Below, $\|\cdot\|$ denotes the euclidean norm.

$$f^{norm}_{x_i} = \frac{f^*_i}{\|(f^*_{x_1}, \ldots, f^*_{x_m})\|} \tag{3.2}$$

P A very impactful attack was proposed by Panchenko et al. in [19]. It was one of the first attacks to consider also some coarse-grained features in addition to packet histograms. In particular, information about overall bandwidth, count of distinct packet sizes, fraction of upstream/downstream packets and bursts are added to the feature set. Additionally, they employ a SVM for classification. Altogether, their attack reaches very high accuracy (more than 96% for 775 websites) even for high class (or, website) counts and has been found robust to many countermeasures [9].

VNG++ In their work in [9], Dyer et al. not only present a comprehensive survey on existing WF attacks and defenses, but also explore the feasibility of website fingerprinting without using a packet histogram, as all previous attacks had been doing. They find that, when using only the overall upload and download bandwidth, the total transmission time and a burst size histogram, results similar to that of the P attack can be achieved while using a Naive Bayes classifier with KDE. Dyer et al. also note that, using the same feature set as the P attack, but the NB classifier instead of a SVM, performance doesn't degrade much from the original P attack, suggesting that using a complex classification algorithm isn't necessary to obtain good results in WF [9].

k-NN Wang et al. [27] further developed the idea of using a simple classification algorithm by resorting to the k-NN classifier and choosing a large set of different features, including total bandwidth and transmission time, occurrence of unique packet sizes, information on packet ordering and bursts (although with a slightly different definition than the one that was given earlier) as well as the size and direction of the first 20 packets in each trace. Furthermore, they determine weights for all features signifying their relative importance, which are multiplied with the feature vectors prior to classification. For the detailed description of the algorithms, refer to [27].

CUMUL A more recent contribution of Panchenko et al. [18], the CUMUL attack also uses a SVM classifier, like the P attack did before. While Wang et al. used a multitude of manually selected features in their k-NN attack (up to 4000), Panchenko et al. focus on creating an abstract representation that implicitly covers the characteristics of a page load [27, 18]. In addition to the basic features of packet count and bandwidth per direction, they create a cumulative sum of the packet sequence (with outgoing packets having negative size) and sample a fixed number n of additional features from this cumulative sum using linear interpolation. In their work, Panchenko et al. find $n = 100$ to yield good results, leading to only 104 features in total [18].

Recent Attacks Many more attacks have been published in recent years which are mentioned here for the sake of completeness. Hayes and Danezis published the k-FP (or k-Fingerprinting) attack where they are using a modified Random Forest classifier to extract an abstract fingerprint for each website, which is then classified using the k-NN classifier [12]. Sirinam et al. employ deep learning techniques, namely Convolutional Neural Networks (CNN), to website fingerprinting, yielding the DF (Deep Fingerprinting) attack [24]. Later, Wang et al. proposed the Adaptive Fingerprinting (AF) attack, greatly reducing the amount of data needed to train the deep learning algorithms[6]. At the same

[6]Usually, deep learning algorithms require a very large amount of training data due to their complex nature and slow optimization convergence

Name	**Type**	**Description**
Linear Padding	deterministic	pad to the next multiple of 128 bytes, or the MTU
Exponential Padding	deterministic	pad to the next power of two, or the MTU
Mice-Elephants	deterministic	pad to 128 bytes if the packet has $l \leq 128$, or otherwise to the MTU
Pad to MTU	deterministic	Each packet size is increased to the MTU
Session Random 255	probabilistic	Sample a value $r \in \{0, 8, 16, ..., 248\}$ uniformly at random for the session. Increase each packet length by this amount.
Packet Random 255	probabilistic	Sample a value $r \in \{0, 8, 16, ..., 248\}$ uniformly at random for each packet. Increase the packet length by this amount.

Table 3.1: An overview of classical padding schemes that have been applied to website fingerprinting [9]

time, AF reduces the need to frequently update the training data due to the sensitivity to content changes of contained websites [26].

3.2 Website Fingerprinting Defenses

In the following, a number of major WF defenses from all the three classes (cf. Section 2.1.3) will be briefly presented.

Classical Padding Defenses There are a multitude of padding schemes that can be applied to the WF setting, some of which are deterministic and some probabilistic. A good overview of them is given in [9]. Table 3.1 also shows a selection of them.

Defense by Noise Several defenses using noise to disguise a website's identifying patterns have been proposed in the past. One notable examples are *Decoy Pages*, as proposed by Panchenko et al. in [19], where a random webpage is loaded in parallel to the desired web page, thus producing noise through unrelated traffic. Another such defense is *Randomized Pipelining*, as deployed in the Tor Browser, which randomizes the order of parallel HTTP requests sent over the same connection (HTTP pipelining) [21, 5].

Name	Authors	Description
BuFLO	Dyer et al. [9]	Sends packets of fixed size d at a fixed interval ρ for at least a fixed time τ, introducing dummy packets when needed
Tamaraw	Cai et al. [4]	Sends packets of fixed size d at fixed intervals, while the interval ρ_{out} for outgoing packets and ρ_{in} for incoming packets are distinct and $\rho_{out} > \rho_{in}$. Pads the number of packets in either direction by multiples of a parameter. [5]
CS-BuFLO	Cai et al. [3]	Similar to BuFLO, but adapts ρ to the available network bandwidth dynamically.

Table 3.2: Examples for morphing defenses trying to reduce information available to the attacker

Morphing Defenses One of the first publications studying morphing defenses is the *Traffic Morphing* algorithm by Wright et al. [28]. This defense tries to make web pages look like other pages by learning a so-called *morphing matrix* that transforms the packet size distribution (*source distribution*) such that it resembles the distribution of a different web page (*target distribution*). The morphing matrix is learned using convex optimization to maximize the similarity between the source and target distributions while keeping the overhead minimal. During the process, packets may be split and resized to match the target distribution best. Another similar algorithm presented by Dyer et al. in [9] is called *Direct Target Sampling*, where the complex morphing step is skipped and target packet sizes are sampled directly from the selected target distribution.

Another class of morphing defenses aims at reducing differences between the traces of different websites, thus making it harder to distinguish them. They try to limit information available to the attacker. Examples for such defenses are shown in Table 3.2.

3.3 Security Bound Estimation According to Cherubin [5]

When evaluating WF attacks and defenses, the traditional approach is to empirically measure their performance on a previously collected data set of packet traces. However, such empirical evaluations are susceptible to noise and fail to produce provable statements on the security of a certain defense. While a defense may perform particularly well against the attacks it is evaluated on, it may just as well horribly fail on a different attack that wasn't considered yet. To address this problem and further advance research on provable security evaluation of WF defenses, Cherubin introduces a novel method centered around the Bayes error in [5].

The *Bayes error* is defined as the minimum probability of error that any classifier can commit given a joint probability distribution on the features and class labels. Intuitively speaking, the Bayes error corresponds to the area of uncertainty in the probability distribution, i.e. the area where the conditional probabilities $p(x \mid c_i)$ and $p(x \mid c_j)$ overlap[7] for any two classes $c_i \neq c_j$ and features x.

To formalize, let $\mathcal{X}$ be the feature space and $\mathcal{C}$ the set of all class labels. Let $p(x, c)$ be a joint probability distribution on $\mathcal{X} \times \mathcal{C}$. Then for the set of classifiers $\mathcal{F} = \{f \mid f : \mathcal{X} \to \mathcal{C}\}$ with R_f being the error of a classifier $f \in \mathcal{F}$ according to p, the Bayes error R^* is defined as

$$R^* = \min_{f \in \mathcal{F}} R_f \tag{3.3}$$

This minimum error is achieved by the *Bayes classifier* $f^* \in \mathcal{F}$ that assigns to each example x the class label c maximizing the probability $p(c \mid x)$ [8].

$$f^*(x) = \arg\max_{c \in \mathcal{C}} p(c \mid x) \tag{3.4}$$

$$R^* = R_{f^*} \tag{3.5}$$

As the true joint probability distribution $p(x, c)$ is typically not known in practice, the exact value of the Bayes error can't be calculated. However, there are well-known methods to estimate a lower bound for the Bayes error. Cherubin uses an estimate based on the NN classifier (see Section 2.1.2) first presented by Cover and Hart [7] deriving a lower bound $\widehat{R}^*$ for the Bayes error R^*

$$\widehat{R}^* = \frac{L-1}{L}\left(1 - \sqrt{1 - \frac{L}{L-1}\widehat{R}_{NN}}\right) \leq R^* \tag{3.6}$$

where $\widehat{R}_{NN}$ is the empirical error of the NN classifier on a data set containing $L = |\mathcal{C}|$ classes.

[7]That is, both probabilities are non-zero

4 Experimental Methodology

4.1 Trace Data

In this work, the data set collected by Liberatore and Levine in [15], called the *Liberatore data set* in the following, was used for all experiments[8]. The data set contains 2000 distinct web sites sampled from the university department's internet traffic and 205 traces per website.

Although Herrmann et al. [13] also published a trace data set that was subsequently used in multiple publications (cf. [19, 9]) and is considered to be of higher overall quality [9], it was no longer accessible under the URL given in their paper, so it couldn't be used here. There were several preprocessing steps done to achieve better data quality. In the following sections, an overview of the main issues in the data set (Section 4.1.1) is given and the preprocessing (Section 4.1.2) is described.

4.1.1 Data Quality

Several issues have become apparent during the use of the Liberatore data set, which demanded careful preprocessing to clean as much inherently inconsistent data as possible. First of all, there were many traces that didn't contain a single packet in either direction. Some websites even didn't have a single trace that contained a packet. Furthermore, some traces only consisted of a single 52 byte packet, which corresponds to a single `ACK`. Such traces clearly don't represent a properly recorded page load and are thus counterproductive to the classification task that is website fingerprinting. According to Dyer et al. [9], 13.8% of the traces in the Liberatore data set are having 10 packets or less in both directions combined, which is highly improbable for most websites. An overview of other statistical properties of the data set is shown in Table 4.1.

[8]The traces are publicly available under `http://traces.cs.umass.edu/`

Traces with 0 packets in one direction	3.1%
Traces with ≤ 5 bidirectional packets	5.2%
Traces with ≤ 10 bidirectional packets	13.8%
Traces with ≤ 1 s duration	29.4%
Median trace duration	2.4 s
Median bidirectional packet count	106
Median overall bandwidth utilization	78,382 byte

Table 4.1: Statistical properties of the Liberatore dataset [15], taken from [9]

Site #1156	**Incoming**		**Outgoing**	
Trace ID	**Bandwidth**	**Packets**	**Bandwidth**	**Packets**
65	68,052 byte	81	8,140 byte	19
66	68,256 byte	84	8,140 byte	19
67	5,968 byte	8	1,312 byte	4
68	67,724 byte	75	8,596 byte	17
69	67,880 byte	78	8,140 byte	19

Table 4.2: An outlier in trace data. Data for web page #1156 is shown with traces #65-#69. Trace #67 clearly deviates from the majority of other traces. For the data shown, `ACK` packets (all packets sized 52 byte) were filtered.

Another irregularity that was observed came with the fact that some traces have an incoming packet as the first one recorded. This behavior is entirely unexpected in the context of loading a web page, as the client first needs to make a HTTP request before the server will send any data back.

Furthermore, there were some clear outliers in the trace data that probably correspond to incomplete page loads. Such outliers may negatively affect classification accuracy. An example for such an outlier is shown in Table 4.2.

4.1.2 Preprocessing

To eliminate as much anomalies as possible before running evaluations on the data, it was analyzed thoroughly using outlier detection to filter out pages with a high fraction of degenerate traces as described in the previous section. Three separate outlier detection methods have been combined to form an ensemble outlier detector, which serves to draw conclusions on possible outliers with a higher certainty than when using only a single detection method. Prior to outlier detection, simple ACK packets (all packets sized 52 byte) were filtered from all traces to reduce noise and focus on actual payload transmitted over the network. Then, traces were aggregated to obtain the utilized bandwidth and packet count in either direction, which yields four dimensions per trace. For outlier detection, three detectors implemented in the `scikit-learn`[9] package [20] were used, namely the `EllipticEnvelope`, `LocalOutlierFactor` and `IsolationForest` algorithms[10]. The output of these detectors is either 1 or -1, marking an inlier or outlier, respectively. Let $o_{env}, o_{lof}, o_{ilf} \in \{-1, 1\}$ be the output of the three detectors. Then, an outlier score o can be defined as

$$o = o_{env} + o_{lof} + o_{ilf}$$

o can only take values $\{-3, -1, 1, 3\}$, which is then mapped to the outlier decision o^* as follows

$$o^* = \begin{cases} \text{inlier}, & o = 3, \\ \text{probable inlier}, & o = 1, \\ \text{probable outlier}, & o = -1, \\ \text{outlier}, & o = -3. \end{cases}$$

This evaluation leads to the conclusion that roughly 11.9% of the traces in the data set are outliers or probable outliers with respect to the previous detection method. The distribution of the count of traces per website for which $o^* \in \{\text{outlier}, \text{probable outlier}\}$ has the properties shown in Table 4.3.

After analyzing the outlier traces, the data set was filtered using two criteria:

1. Drop all websites having 29 or more outlier traces, thus keeping the 75% percentile of the data in regard to outlier count.
2. From the remaining websites drop all that have at least one trace with three or fewer packets in both directions. Traces that have three or fewer packets in one direction, but more than three in the other one are not considered.

[9]`https://scikit-learn.org/`

[10]For details on the algorithms, please refer to `https://scikit-learn.org/stable/modules/outlier_detection.html`

Mean	24.32
Standard Deviation	8.80
Minimum	0
25% Percentile	19
Median	24
75% Percentile	29
Maximum	63

Table 4.3: The distribution of outlier or probable outlier count per website

After these preprocessing steps, 532 websites out of the initial 2000 remain in the data set.

4.2 Evaluating Attack Performance

Attacks and defenses were evaluated against each other using a generic framework implementing a general-purpose evaluation pipeline implemented in the course of this thesis. The pipeline consists of four stages, which are briefly described in the following sections. All implementations were done in Python 3.8 using the `scikit-learn` machine learning toolkit [20], as well as the `numpy`[11] [11] and `scipy`[12] [25] libraries for scientific computing and `pandas`[13] [17, 22] for data manipulation and I/O. APIs have been modeled after the example of `scikit-learn` [2] where possible.

An instance of the pipeline is always ran with a set of training and a set of test data, which are distinct. It may run multiple evaluations for different attacks and defenses in parallel, for which the `multiprocessing` module from Python's standard library is utilized. The number of training and test examples to use for evaluation as well as the offset between training and test data can be customized. For all experiments, the offset between training and test data was set to 0, i.e. both sets are read from a single contiguous range of trace IDs. The first trace id of this set was randomized. Varying numbers of training examples had to be used, following the previous author's suggestions. See Table 4.4 for the exact numbers. The number of test examples per website was always set to be 4. For every pair of defense and attack, the evaluation was run 10 times.

[11]`https://numpy.org/`
[12]`https://www.scipy.org/`
[13]`https://pandas.pydata.org/`

Attack	Training Examples
LL [15]	4
H [13]	4
P [19]	18
VNG++ [9]	16*

Table 4.4: The number of training examples used for each attack
* = For VNG++, Dyer et al. did not specify the number they used in their publication, therefore the default setting from their published source code was used.

4.2.1 Data Set

The data set stage takes care of loading traces and abstracts from the storage technology used for the data set. It mainly consists of an implementation of the abstract class `Dataset` encapsulating all logic required to load a dataset and transform it into a `pandas` data frame with the following columns, where each row corresponds to a recorded packet:

- `site_id`: The ID of the website, must be unique within the data set
- `trace_id`: The ID of the packet trace, must be unique within the same website
- `collection_time`: The time at which the trace collection started
- `time`: The offset of the packet's arrival from the first packet in the trace in milliseconds
- `size`: The length of the packet, negative for outgoing packets and positive for incoming packets

An implementation is provided for the Liberatore dataset, containing additional functionality to download, extract and parse the PCAP logs of the data set and store them in the HDF5 format for quick and easy access.

4.2.2 Defense

After the data set follows the optional defense stage, which is only present when attacks should be evaluated against defenses. This stage is implemented using the `Defense` class, which takes packet traces in the format as returned by `Dataset` and returns the same data structure. Defenses normally change the `size` and/or `time` columns of a trace.

Implementations of all padding defenses mentioned in Section 3.2 are provided, as well as Gaussian padding in packet-random and session-random mode.

4.2.3 Feature Extraction

In the feature extraction stage, packet traces are transformed to a single vector of features and a class label (the `site_id`). The output of this stage is no longer expected to be a `pandas` dataframe, but a pair of `numpy` arrays, the first containing all feature values and the second containing the class labels.

Individual feature sets can be combined using the features from both sets and transformed, using any `scikit-learn` transformer, as done with the TF transformation and cosine normalization in the attack by Herrmann et al. [13].

The framework created during this thesis provides implementations for the feature sets of Liberatore and Levine [15], Herrmann et al. [13], Panchenko et al. [19, 18] and Dyer et al. [9]. Most operations have been implemented solely as `numpy` array operations and `pandas` aggregations due to the superior speed because of them running in native C code [11, 17].

4.2.4 Classifier

The final stage of the pipeline is a classifier which is trained using the features and class labels extracted from the training data. Then, the test data's features are classified with the trained classifier and the chosen performance measure is calculated using the predicted classes against the actual class labels. The default and most common choice for said performance measure is accuracy (the fraction of correctly classified examples).

All classification algorithms have been implemented using `scikit-learn`. For Multinomial Naive Bayes and SVM the implementations provided by the library were used. Unfortunately, it doesn't contain an implementation of Naive Bayes using Kernel Density Estimation, so that variant was implemented following `scikit-learn`'s typical API design [2]. Also, an efficient variant of one-dimensional KDE was implemented following the algorithm from the WEKA Java machine learning toolkit[14] [10], which was used by most prior publications that were using the NB classifier with KDE (cf. [15, 9]).

[14] `https://www.cs.waikato.ac.nz/ml/weka/`

4.3 Error Bound Estimation

When estimating lower bounds for classifier error according to the method developed by Cherubin [5], the general flow and pipeline structure resembles closely the one described in the previous section. However, as the final stage of the pipeline, the attack's classifier is substituted by Cherubin's error estimator. Also, there are no distinct training and test sets in this pipeline as the estimator calculates the accuracy of the NN classifier using 5-fold cross validation (CV). In this method, the data set is split up into five equal parts, of which each one serves as the test set once while the other four become the training set. After measuring the classification accuracy for all five runs, the results are averaged to obtain a final score.

For this evaluation, 100 websites were used with 100 traces each which were loaded from a contiguous range of trace IDs. The first trace ID of the data set was randomized. The experiment was repeated 50 times for both, the closed-world and open-world mode (see below).

4.3.1 Bayes Error Estimator

The Bayes error estimator operates either in closed-world or open-world mode. In closed-world, the data set received is directly split up for cross validation and accuracy is measured using all websites and traces. In contrast, when using the open-world mode, for each website present in the data, all traces of that website and exactly as many randomly selected traces of different websites are passed on to the cross validation. This corresponds to the One-vs-All mode described by Cherubin [5], in which the monitoring of a single website against all others is simulated.

After calculating the CV accuracy, the formula from Equation 3.6 is used to calculate $\widehat{R}^*$ as the lower bound of the Bayes error.

5 Results

5.1 Empirical Performance of Gaussian Padding

5.2 Estimated Security Bounds

6 Discussion

7 Conclusion

Bibliography

[1] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A training algorithm for optimal margin classifiers”. In: *Proceedings of the fifth annual workshop on Computational learning theory - COLT ’92*. ACM Press, 1992. DOI: 10.1145/130385.130401.

[2] Lars Buitinck et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[3] Xiang Cai, Rishab Nithyanand, and Rob Johnson. “CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense”. In: *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. ACM, Nov. 2014. DOI: 10.1145/2665943.2665949.

[4] Xiang Cai et al. “A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Nov. 2014. DOI: 10.1145/2660267.2660362.

[5] Giovanni Cherubin. “Bayes, not Naïve: Security Bounds on Website Fingerprinting Defenses”. In: *Proceedings on Privacy Enhancing Technologies* 2017.4 (Oct. 2017), pp. 215–231. DOI: 10.1515/popets-2017-0046. URL: https://petsymposium.org/2017/papers/issue4/paper50-2017-4-source.pdf.

[6] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. DOI: 10.1007/bf00994018.

[7] T. Cover and P. Hart. “Nearest neighbor pattern classification”. In: *IEEE Transactions on Information Theory* 13.1 (Jan. 1967), pp. 21–27. DOI: 10.1109/tit.1967.1053964.

[8] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley John + Sons, Oct. 1, 2000. ISBN: 0471056693. URL: https://www.ebook.de/de/product/3244086/richard_o_duda_peter_e_hart_david_g_stork_pattern_classification.html.

[9] Kevin P. Dyer et al. “Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012. DOI: 10.1109/sp.2012.28. URL: https://cise.ufl.edu/~teshrim/tmAnotherLook.pdf.

[10] Eibe Frank, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench. Online Appendix for ”Data Mining: Practical Machine Learning Tools and Techniques”, Morgan Kaufmann, Fourth Edition*. 2016. URL: https://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pdf (visited on 08/18/2021).

[11] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[12] Jamie Hayes and George Danezis. “*k*-fingerprinting: A Robust Scalable Website Fingerprinting Technique”. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC’16. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 1187–1203. ISBN: 9781931971324. URL: https://nymity.ch/tor-dns/pdf/Hayes2016a.pdf.

[13] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. “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*. ACM Press, 2009. DOI: 10.1145/1655008.1655013. URL: https://dl.acm.org/doi/pdf/10.1145/1655008.1655013.

[14] Andreas Hülsing, Tanja Lange, and Kit Smeets. “Rounded Gaussians: Fast and Secure Constant-Time Sampling for Lattice-Based Crypto”. In: *Public-Key Cryptography – PKC 2018*. Springer International Publishing, 2018, pp. 728–757. DOI: 10.1007/978-3-319-76581-5_25.

[15] Marc Liberatore and Brian Neil Levine. “Inferring the Source of Encrypted HTTP Connections”. In: *Proceedings of the 13th ACM conference on Computer and communications security - CCS ’06*. ACM Press, 2006. DOI: 10.1145/1180405.1180437. URL: https://dl.acm.org/doi/pdf/10.1145/1180405.1180437.

[16] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Pr., Sept. 1, 2008. 496 pp. ISBN: 0521865719. URL: https://www.ebook.de/de/product/7455223/christopher_d_manning_prabhakar_raghavan_hinrich_schuetze_introduction_to_information_retrieval.html.

[17] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. SciPy, 2010. DOI: 10.25080/majora-92bf1922-00a.

[18] Andriy Panchenko et al. “Website Fingerprinting at Internet Scale”. In: *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016. DOI: 10.14722/ndss.2016.23477.

[19] Andriy Panchenko et al. “Website Fingerprinting in Onion Routing Based Anonymization Networks”. In: *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society - WPES ’11*. ACM Press, 2011. DOI: 10.1145/2046556.2046570. URL: https://dl.acm.org/doi/pdf/10.1145/2046556.2046570.

[20] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[21] Mike Perry. *Experimental Defense for Website Traffic Fingerprinting*. Sept. 4, 2011. URL: https://blog.torproject.org/experimental-defense-website-traffic-fingerprinting (visited on 08/11/2021).

[22] Jeff Reback et al. *pandas-dev/pandas: Pandas 1.3.2*. 2021. DOI: 10.5281/ZENODO.3509134.

[23] scikit-learn developers. *Plot the support vectors in LinearSVC*. 2020. URL: https://scikit-learn.org/stable/auto_examples/svm/plot_linearsvc_support_vectors.html (visited on 08/13/2021).

[24] Payap Sirinam et al. “Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Jan. 2018. DOI: 10.1145/3243734.3243768.

[25] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (Feb. 2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[26] Chenggang Wang et al. “Adaptive Fingerprinting: Website Fingerprinting over Few Encrypted Traffic”. In: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. ACM, Apr. 2021. DOI: 10.1145/3422337.3447835.

[27] Tao Wang et al. “Effective Attacks and Provable Defenses for Website Fingerprinting”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 143–157. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang_tao.

[28] Charles Wright, Scott Coull, and Fabian Monrose. “Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis”. In: *Proceedings of the 16th annual Network and Distributed System Security Symposium - NDSS ’09*. NDSS, Jan. 2009. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/wright.pdf.

List of Figures

List of Tables