NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEFENDING AGAINST DEEP LEARNING-BASED VIDEO FINGERPRINTING ATTACKS WITH ADVERSARIAL EXAMPLES

by

Blake A. Hayden

June 2022

Thesis Advisor:	Armon C. Barton
Second Reader:	Joshua A. Kroll

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
DEFENDING AGAINST DEEP LEARNING-BASED VIDEO FINGERPRINTING ATTACKS WITH ADVERSARIAL EXAMPLES

5. FUNDING NUMBERS

6. AUTHOR(S) Blake A. Hayden

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

In an increasingly digital world, online anonymity and privacy is a paramount issue for internet users. Tools like The Onion Router (Tor) offer users anonymous internet browsing. Recently, however, Tor's anonymity has been compromised through fingerprinting, where machine learning models are used to analyze Tor traffic and predict user viewing habits, with some models achieving an accuracy of over 99%. There are defenses for Tor that attempt to prevent fingerprinting, but many are defeated by new techniques that utilize Deep Neural Networks (DNNs). New defenses that are robust against DNNs use adversarial examples to fool the classifier, but those defenses either assume the user has access to the full traffic trace beforehand or require expensive maintenance from Tor servers. In this thesis, we propose Prism, a defense against fingerprinting attacks that uses adversarial examples to fool classifiers in real time. We describe a novel method of adversarial example generation that enables adversarial example creation as input is learned over time. Prism injects these adversarial examples into the Tor traffic stream to prevent DNNs from accurately predicting sites that a user is viewing, even if the DNN is hardened by adversarial training. We show that Prism reduces the accuracy of defended fingerprinting models from over 98% to 0%. We also show that Prism can be implemented entirely on the server side, increasing deployability for users who run Tor on devices without GPUs.

14. SUBJECT TERMS adversarial examples, defending Tor, website fingerprinting, video fingerprinting, deep fingerprinting, defending anonymity, anonymity	**15. NUMBER OF PAGES** 59
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DEFENDING AGAINST DEEP LEARNING-BASED VIDEO FINGERPRINTING ATTACKS WITH ADVERSARIAL EXAMPLES

Blake A. Hayden
Ensign, United States Navy
BS, United States Naval Academy, 2021

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2022

Approved by: Armon C. Barton
Advisor

Joshua A. Kroll
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In an increasingly digital world, online anonymity and privacy is a paramount issue for internet users. Tools like The Onion Router (Tor) offer users anonymous internet browsing. Recently, however, Tor's anonymity has been compromised through fingerprinting, where machine learning models are used to analyze Tor traffic and predict user viewing habits, with some models achieving an accuracy of over 99%. There are defenses for Tor that attempt to prevent fingerprinting, but many are defeated by new techniques that utilize Deep Neural Networks (DNNs). New defenses that are robust against DNNs use adversarial examples to fool the classifier, but those defenses either assume the user has access to the full traffic trace beforehand or require expensive maintenance from Tor servers. In this thesis, we propose Prism, a defense against fingerprinting attacks that uses adversarial examples to fool classifiers in real time. We describe a novel method of adversarial example generation that enables adversarial example creation as input is learned over time. Prism injects these adversarial examples into the Tor traffic stream to prevent DNNs from accurately predicting sites that a user is viewing, even if the DNN is hardened by adversarial training. We show that Prism reduces the accuracy of defended fingerprinting models from over 98% to 0%. We also show that Prism can be implemented entirely on the server side, increasing deployability for users who run Tor on devices without GPUs.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

BWO bandwidth overhead

CL convolutional layer

CNN Convolutional Neural Network

DF Deep Fingerprinting

DNN Deep Neural Network

ML machine learning

PGD projected gradient descent

Tor The Onion Router

VF video fingerprinting

WF website fingerprinting

WT Walkie-Talkie

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Introduction

Privacy on the web is a hot topic both technologically and politically. Some users simply want to browse the web anonymously for peace of mind knowing that no third party is able to track what they are doing. For others, however, like those living under a regime that engages in censorship, online anonymity becomes a necessity. Programs, such as The Onion Router (Tor), have created a way for users to browse the web anonymously [1]. The anonymity offered by Tor is robust, but it did not take long for adversaries to discover ways to circumvent Tor's privacy and identify users' browsing habits.

The most recent development in the field of attacking Tor has been using Deep Neural Networks (DNNs) to analyze first hop network traffic and predict what website the victim is visiting. This attack is known as website fingerprinting (WF). More specifically, using a DNN to perform WF is called Deep Fingerprinting (DF) and was demonstrated in Sirinam et al's work [2]. Another way for an attacker to learn what a victim is viewing on the web is through video fingerprinting (VF). This technique works similarly to WF, but the DNN is used to analyze the traffic and determine which video a user is streaming rather than which website he is viewing.

The objective of this thesis is to develop an effective defense against both VF and WF attacks that are utilizing a DNN classifier. There have been several papers that explore possible defenses against WF attacks, such as Mockingbird [3], Walkie-Talkie [4], and *Dolos* [5], but no defenses have been created for defending against VF attacks. Additionally, many of the defenses that have been proposed for WF do not model the defense in a real-world scenario where the information being defended is generated in real-time. The defenses are therefore not implementable as a defense for Tor. This idea is discussed more in Section 2.4.

In this paper, we propose *Prism*, a VF and WF defense that uses adversarial examples to defend Tor traces in real-time. We accomplish this by utilizing a novel method of adversarial example generation which implements a modified version of projected gradient

descent (PGD) to craft adversarial examples on input that is learned over time. *Prism* uses this approach to craft adversarial examples capable of fooling the classifier when injected into the traffic stream. We also modify DF to create a VF classifier to test our defense against both VF and WF classifiers.

By implementing our *Prism* defense, we show that we can effectively defeat fingerprinting, significantly reducing the accuracy of DF models trained to perform VF and WF. Specifically, when we apply the *Prism* defense to traces captured on Tor, we show that *Prism* reduces the accuracy of undefended VF models from 92% down to 0% and reduces the accuracy of undefended WF models from 98% down to 0%. We also test *Prism* with defended models, which are hardened to resist adversarial examples, and show that *Prism* still defeats the classifiers, reducing the defended VF model accuracy to 0% and the defended WF model accuracy to 56%.

1.2 Thesis Organization

This thesis is organized into the following chapters:

- **Background and Related Work** where we explain the technical background regarding the techniques employed in the paper and analyze current approaches to defending against fingerprinting attacks.
- ***Prism* Design** where we describe the design of the *Prism* defense and explain how the defense is implemented algorithmically.
- **Evaluation** where we present and analyze the results of the *Prism* defense when tested against various fingerprinting models.
- **Discussion** where we discuss possible implementation decisions, limitations of the defense, and possible future work to improve the defense.
- **Conclusion** where we summarize our findings in the paper.

CHAPTER 2:
Background and Related Work

2.1 Tor Onion Router

Tor is an internet browser and routing protocol that was specifically designed to protect users' online anonymity. To accomplish this, Tor layers encryption on the network packet before sending it, so the original source *and* destination cannot be discovered by intercepting the packet. Before sending the packet, Tor randomly selects a *circuit* between the client and destination consisting of three relay nodes [6]. The circuit is chosen at random and will be used for 10 minutes at which point a new circuit will be selected for user [7]. As the packet is forwarded between the nodes, each node will strip off a layer of encryption to reveal the next hop in the circuit and forward it on until the packet exits the Tor network and arrives at the destination. Figure 2.1 shows a graphical representation of how packet forwarding is accomplished over Tor.

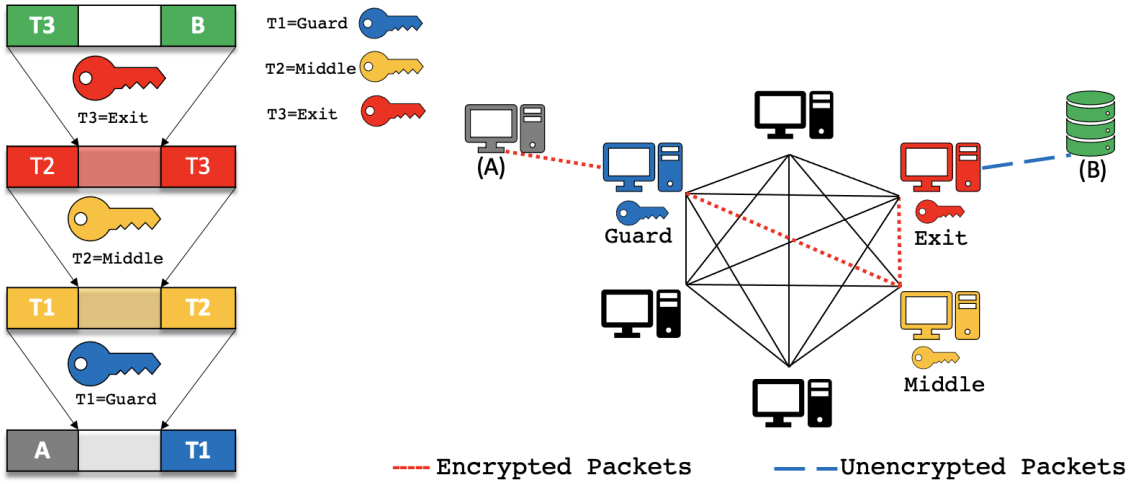

Figure 2.1. Tor network. Source: [7].

In the diagram, client (A) wants to reach destination (B). First, the client selects a *circuit* which will tunnel the packet through the network. The circuit consists of a *guard* node, *middle* node, and *exit* node. When a new circuit is selected, only the *middle* and *exit* nodes are changed; the *guard* node is only changed if it is no longer available or a period of 60 days to 9 months is reached [6]. After selecting the circuit, the client will encrypt the message three times using three different keys, each corresponding to one of the nodes in the circuit. The client then forwards the encrypted packet to the *guard* node which decrypts the first layer of encryption. After decrypting, the guard node will know the address of the *middle* node and forward the packet on. The *middle* and *exit* nodes repeat this process and deliver the packet to the destination. Note that no party in this system besides the client knows the source and destination, only the previous and following hop.

The layered encryption approach of Tor significantly increases latency for users. [6] proposed a method to intelligently select Tor circuits that will decrease latency. Additionally, it is possible for nodes in the Tor network to become compromised which can lead to a compromise in user anonymity. [8] outlines a method of selecting destination independent Tor circuits to improve security by avoiding nodes which may be compromised.

Researchers have also found ways to attack anonymity on Tor by using a malicious guard node to perform *fingerprinting* attacks (discussed in Section 2.3) on the traffic from users and determine the sites they are visiting [9]. Reference [10] addresses this by creating *guard sets* which allow a user to select a guard node from a set of secure guard nodes to limit the ability of malicious guards to inject themselves into the Tor network. It is still possible, however, for an adversary acting in the role of an Internet Service Provider to observe the encrypted traffic and perform fingerprinting.

2.2 Machine Learning

The attacks that perform WF have relied on the science of machine learning (ML). ML can be defined as, "a field of study that gives computers the ability to learn without being explicitly programmed" [11]. *Neural networks* are a method within ML that are useful when a programmer wants to model and predict the outcomes of a complicated data set. Neural Networks accomplish this by analyzing a set of specified *features* in the data set and learning relations between the input x and output y to model a function f, such that $y = f(x)$.

Neural networks have layers of *neurons* where each neuron can be either 1 or 0. The resulting structure of layers are referred to as a *model*. Each neuron is connected to every neuron in the following layer by a *weight*, w. As information travels through the model, each neuron computes the sum of the inputs multiplied by the weights, so for n inputs,

$$z = \sum_{i=0}^{n} x_i * w_i \qquad [11]$$

The sum z is then fed into an *activation function* which becomes the value of the neuron [11]. During the training phase of the model, the *weights* are adjusted using a *loss function* to minimize the loss between output y and desired output $\hat{y}$.

As the data set complexity increases, the model must be expanded as well to identify more subtle relationships within the data. This can be done by adding more layers to the model. As the model's complexity grows, however, it requires more training data, with larger models requiring millions of samples of data to successfully train.

2.2.1 Deep Neural Networks

One type of neural network is called a DNN. A DNN can get very large, often having more than 10 layers with each layer having hundreds of neurons [11]. These models excel at performing subtle classifications on large data sets, and have seen extensive use in image classification [11]. One advancement that has tremendously improved the performance of DNNs is the use of *convolutional layers (CLs)*. When a DNN makes use of CLs, it is referred to as a Convolutional Neural Network (CNN).

CNNs are a way for the model to abstract the input so it becomes easier to identify less obvious patterns within the data. In image classification, CNNs work by focusing on a smaller section of pixels within the image called a *receptive field*. Instead of mapping each pixel to the neurons above, a CNN looks at a small rectangle within the image and computes a neuron value based on that selection [11]. This enables the CNN to focus on pulling features out of different sections of the image to improve classification. While in this example the CNN is used for image classification, CNNs can be used in any other model where less obvious features are important to successfully classify the input.

2.3 Website Fingerprinting

In website fingerprinting, a local adversary listens to network traffic to and from a client that is visiting websites and attempts to identify which website the client is viewing [12]. When performing WF, the attack will be classified as being either closed-world or open-world. Closed-world attacks are performed in a laboratory setting where the visited websites are pulled from a predetermined set and there are not unanticipated network packets serving as noise. Conversely, open-world attacks are performed in the wild where the attacker has no prior knowledge of which websites the client is viewing or what type of packets may be sent.

Anonymous browsers like Tor make WF attacks much more difficult by encrypting the entire network packet, so the only information that an attacker can gather is the direction, timestamp, and size associated with each packet. Before machine learning capabilities advanced to the level they are now, WF attacks over Tor in an open-world setting based only on these features were not feasible. With advents in ML, however, WF attacks over Tor are now possible and have been thoroughly documented.

2.3.1 Deep Fingerprinting

Deep Fingerprinting (DF) is a new model that uses a Deep Neural Network to perform WF [2]. The convolutional layers within the DNN assist the model in pulling out subtle features within the data set which greatly improves the model's accuracy when used on traffic traces that have been defended. One weakness that most defenses have is that an attacker can include defended traffic traces in the training set so the classifier can learn the defense and circumvent it when classifying traces.

DF demonstrates this with a defense called WTF-PAD [13], a WF defense that has been adopted by Tor. WTF-PAD works by sending bursts of dummy packets whenever there is a large delay between packet arrival times, and this defense reduced the accuracy of some WF models from 92% down to 17% with a bandwidth overhead (BWO) of 60% [13]. By including WTF-PAD defended traffic in their training set, the DF model was able to achieve a closed-world accuracy of over 90%, essentially breaking the WTF-PAD defense [2]. It is important to note that in an open-world evaluation of the DF model on the Walkie-Talkie (WT) defense, the model could only achieve an accuracy of approximately 35%. This

is largely the result of the burst modeling which is specifically designed to create collisions and reduce the certainty of the model. The WT defense is not implementable in a real-world setting, however, because it requires knowledge of the full trace before crafting a defense.

2.3.2 BurNet

While traditional WF methods require the attacker to be *local*: i.e., have access to both incoming and outgoing packets, [14] proposed *BurNet*, a Convolutional Neural Net (CNN) which is able to perform fine grained WF with unidirectional packets, enabling a *remote* attacker. The work uses the term fine grained WF to refer to identifying specific webpages a user is viewing, as opposed to only the domain. *BurNet* accomplishes this by creating unidirectional packet bursts going from the server to the client. A single burst is collected from a series of packets and computed based on the packet sequencing and encrypted message size [14]. Using bursts reduces the size of the training set and speeds up training while also improving the performance of the model over alternative approaches, especially in a remote attacker scenario.

The dataset used by [14] was built by accessing *www.jd.com* and *www.youtube.com*. The experiments performed on the youtube dataset were in essence performing VF, and *BurNet* identified the correct webpage (video) with 98% accuracy in the local attack scenario, outperforming DF and the other models used. In the paper, it is noted that having access to both direction sequences is preferred and results in higher accuracy, but since the paper was primarily concerned with TLS traffic, it focused on unidirectional bursts. With Tor, however, a remote attacker scenario is not feasible since the header information is encrypted, so an attacker must be local [1].

2.3.3 Video Fingerprinting

Video fingerprinting differs from website fingerprinting because instead of intermittent packet transmissions containing websites, video streaming creates a constant flow of traffic to be analyzed. With the flexibility of machine learning, however, we are able to adjust established WF models and train them on video traces to perform VF. For the purposes of this paper, we use a modified Deep Fingerprinting model to perform VF. Although *BurNet* slightly outperformed the DF in testing, the literature indicates that *BurNet* will be similarly susceptible to our adversarial example attack since it relies on a CNN [15].

2.4 Adversarial Examples

[15] explores a weakness identified in DNNs: adversarial examples. [15] noted that DNNs are sensitive to slight changes in the input image due to reliance on *local* generalization, where the DNN is able to assign a high probability guess to a section of the image that does not contain data similar to training input. For the most part this approach works very well, but it can be exploited to attack the model. Adversarial examples are a way to fool image classifying DNNs by adding small changes to the input image that are imperceptible to the human eye but cause the classifier to misclassify the image with a high degree of certainty. Distance metrics L_0, L_2, and L_∞ are used to quantify how much the image changes, where

- L_0 is the total number of pixels that were changed.
- L_2 is the Euclidean distance between the two images.
- L_∞ is the greatest change to any one pixel in the image. [16]

The method employed by [15] to generate adversarial examples was to add noise to the input image until it is misclassified by the model, then minimize that noise so it changes the input image as little as possible while still fooling the model. An example of adversarial examples is shown in Figure 2.2 (reproduced from [15]).

Figure 2.2. Left shows the original input, middle shows the perturbations that are added to the photo on the left, and right shows the resulting adversarial example image. The images on the left are correctly classified, and the images on the right are misclassified as an ostrich, struthio, and camelus. Source: [15].

Another interesting finding from [15] is that the adversarial examples are highly transferable, meaning those same adversarial example photos can be fed to another model, and that model often misclassifies the photo with the same incorrect prediction class.

Projected Gradient Descent

One way of producing adversarial examples is through PGD. PGD is similar to the Fast Gradient Sign Method, an attack method that calculates the gradient from the loss for a given model prediction, then takes one step down the sign of the gradient to generate an adversarial example [17]. PGD introduces a small amount of noise around the original example, then repeatedly steps down the gradient [16]. A mathematical representation of this approach is shown below in Equation 2.1 (pulled from [17]).

$$x^{t+1} = \Pi_{x+\mathcal{S}}\left(x^t + \alpha \text{sgn}(\nabla_x L(\theta, x, y))\right) \tag{2.1}$$

where L is a loss function, α is a step size, and Π_{x+S} is an l_p-bound on X. It is important to note that PGD assumes the attacker will use the entire input in conjunction with model gradients to craft adversarial examples. This assumption does not hold in a network setting, however, when the full input is not known beforehand.

2.4.1 Defending Against WF Attacks using Adversarial Examples

Mockingbird

In [3], the authors apply adversarial examples to create *Mockingbird*, a defense against WF attacks. In their approach, they attempt to defend a given source trace by randomly selecting a *target sample* from a training set and making gradual changes to the source trace to get closer to the target sample [3]. The defense is finished when a sample WF model misclassifies the source trace. They note that the source trace does not need to be classified as the target sample for the defense to succeed. *Mockingbird* relies on a the degree of confidence from the WF model with the goal being high confidence in regard to the target sample. It adjusts the source sample to maximize the confidence value.

The *Mockingbird* defense successfully reduces the accuracy of a DF classifier to less than 40% with a lower BWO than alternative defenses such as WTF-PAD and WT. *Mockingbird* is not a feasible solution to WF attacks, however, because it requires knowledge of the entire trace before it can craft perturbation to defend that trace. In a real-world scenario, the client does not fully observe the trace until all the content has been downloaded. At that point, the adversary would have also passively observed the full trace and predicted the content before the client could add defensive perturbation. For this reason, *Mockingbird* is considered a proof of concept rather than a realistic defense [3].

Adversarial Patches

Reference [18] proposes a novel method of adversarial patches as a way to attack DNNs. Adversarial patches differ from regular adversarial examples by only affecting a smaller area of the image, or the trace in the case of WF and VF. Reference [5] proposes *Dolos* which uses adversarial patches as a defense for WF, outperforming other methods such as WTF-PAD and *Mockingbird* while being implementable in a real-world setting. In their

work, the authors combine adversarial patches with a user secret to create unique adversarial patches that consistently fool the classifier with only 30% BWO [5].

The real-time patching approach works by selecting a website trace, W, that is dissimilar to the website a user is trying to visit, and combines it with a user secret u to compute a patch with (u, W). In the paper, the authors address deployment by arguing for a database, maintained by Tor, which stores website traces divided into sensitive and non-sensitive sites that is automatically queried when a user attempts to defend their traffic. This approach enables thin clients (clients without GPU access) to defend their traces. A database of this magnitude, however, incurs a significant cost to maintain, and it does not scale well to accommodate VF where traces are much larger and more numerous.

Our *Prism* approach does not require precomputed adversarial examples, so we eliminate the need for database maintenance. We show that *Prism* can be implemented as a gateway only defense, meaning that only the Tor guard injects perturbation packets, eliminating the need for the user to have access to a GPU. Additionally, we discuss possible implementation in Section 5.1.2 that would enable traffic from both the client and the gateway while only computing the defense at the gateway which expands the effectiveness of *Prism* on scaled data for WF. This is a significant step towards deployability because many Tor users prefer to run thin clients and therefore would not benefit from a defense that requires a client side GPU.

2.4.2 Defenses Against Adversarial Examples

Reference [19] discusses possible ways to defend a model against an adversarial example attack. The paper describes a handful of defenses, such as detecting if adversarial examples are present in the input [20], trying to remove the perturbations from the input [21], including adversarial examples in the training set [22], and others. The authors point out that all of the defenses in the paper tend to be effective for defending against different aspects of the attack, but none do a very good job of generalizing, especially to unseen attacks. PadNet is another approach which introduces padding classes to separate possible classes as well as minimizing gradients to prevent adversarial examples from traversing the gradient of the loss function [16].

Some of the defenses, such as the inclusion of adversarial examples in the training set,

are intended to add robustness to the model. This improves model regularization and helps to prevent overfitting, which improves model performance on an unperturbed test set and a single iteration adversarial example attack [19]. The resulting model does not perform better on black box adversarial examples, however, indicating that the hardened model is sensitive to adversarial examples that were developed for a different model.

In this paper we include adversarial examples in the training set as a defense for three main reasons. Firstly, including adversarial examples in the training set is a simple hardening technique that is easy to do, so it is a first step for an attacker to defend their model. Secondly, each approach has significant trade-offs, and none are able to provide an all-encompassing defense against adversarial examples [19]. This approach has shown promising results for improving generalization which we determined was most important in our experimentation. Finally, in a real-world scenario, an attacker collecting traces to train a model for fingerprinting will see traces defended with adversarial examples, so including adversarial examples in the training set emulates a real-world scenario most closely.

CHAPTER 3:
Prism Design

We will now describe the rationale and motivation behind our approach and describe our design in detail.

Our main goal is to provide a means to effectively defend a trace in *real-time*. Other papers have shown possible defenses using adversarial examples, [3], [5], but as stated in Section 2, not many of the effective defenses can be implemented in a real-world scenario. This is because adversarial examples are designed to take full model inputs and make small perturbations to the entire input, which is not suitable in network traffic analysis because a trace cannot be determined before the download is complete.

To solve this issue, we rely on *perturbation windows*. We define a perturbation window as a subsection of the trace that is eligible for perturbation tuning. The window is the last n packets of the trace that have been seen, where n is a defense parameter. The window acts as a pool for the packets before sending them, and perturbs the entire window as a batch before transmitting the packets with the adversarial examples. When perturbing the window, *Prism* uses the information seen previously in the trace, but only makes changes to the current window. Adversarial examples perform better the more information is available about the input, so by including the previous packets when perturbing the current window, the adversarial examples get better as the trace continues. Figure 3.1 depicts the use of the perturbation window.

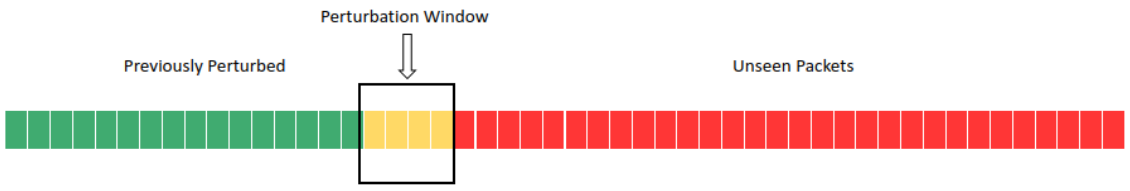

Figure 3.1. Perturbation window representation

3.1 Generating Adversarial Traces

Since we are not able to alter the original packets of the trace, our first step is inserting *perturbation columns* into the dataset which simulates perturbation packets being inserted strategically by the client or guard during a live TCP stream. These perturbation columns are randomly initialized within the min and max value range from the dataset. We want the columns to be evenly distributed throughout the trace so that we do not end up with large groups of dummy packets. This will prevent interruptions to the video stream as well as make it difficult for the adversary to guess where the perturbations are and simply drop that section of the trace before evaluating.

The perturbation columns are entered into the dataset in groups which are spread throughout the trace. To insert perturbation columns, we used values of desired saturation, s, and width, w, where saturation refers to a desired BWO (as a percentage between 0 and 1) and width refers to the number of columns entered in each grouping. We limit BWO to less than 100% because a defense that uses over 100% BWO will incur too much latency to be usable. The first thing we calculate is the step size. The step size is a portion of the original trace in which a group of perturbation columns will be randomly inserted. We want a step size such that if a perturbation group of width w is inserted within step packets, we get a saturation level (additional perturbation packets) of s. Mathematically, we can represent this as

$$\frac{\text{step} + w}{\text{step}} = 1 - s \tag{3.1}$$

We can now solve for step, and get

$$\text{step} = \frac{w}{s} \tag{3.2}$$

In the Python implementation, the *randint* function, which is used to select a random index within the step window, is non-inclusive, so we need to add 1 to account for this. Additionally, *randint* requires integer arguments, so we have to truncate step to an integer, which gives us

$$\text{step} = \lfloor \frac{w}{s} \rfloor + 1 \tag{3.3}$$

An index is randomly selected from this step range for the perturbation column entry. Applying this step value, we can create a trace with perturbation columns distributed evenly throughout with Algorithm 1.

Algorithm 1 Inserting Perturbation Columns

Input

trace	A network trace capture
s	The desired BWO of perturbation columns s.t. $0 < s < 1$
w	The desired width of the perturbation column entries
pad_mask	An array of $\{0,1\}^{\lambda}$ where padding in the trace is 0
μ	The length of the trace

Output

trace	The trace with random perturbation columns inserted
pert_mask	An array of $\{0,1\}^{\lambda}$ where perturbation columns are 1

pert_mask$[\mu] \leftarrow \{0, 0, ..., 0\}$
step $\leftarrow \lfloor \frac{w}{s} \rfloor + 1$
$i \leftarrow 0$
while $i < \mu$ **do**
  index $\leftarrow randint(i, \text{step} + i)$
  $i \leftarrow i + \text{step} + w$
  if index $> \mu$ **then**
    index $\leftarrow \mu - 1$
  end if
  $j \leftarrow 0$
  while $j < w$ **do**
    $r \leftarrow randint(min(\text{trace}), max(\text{trace}))$
    trace.$insert$(index + j, r)
    pert_mask.$insert$(index + j, 1)
    $j \leftarrow j + 1$
  end while
end while
trace $\leftarrow$ trace $*$ pad_mask

This algorithm inserts random integers sampled from the range of inputs seen into the trace in perturbation groupings. It also tracks the locations of the inserts by inserting a 1 in the *pert_mask* which is used when performing the PGD optimization on the trace. Finally, the

trace is multiplied by a *pad_mask* to restore any padding that may have been clobbered. This is necessary because the perturbation columns are added after the data is processed into an array of traces and padded or snipped to a common length. Figure 3.2 shows a simplified depiction of the process described in Algorithm 1 where we set $s = 0.2$ and $w = 2$.

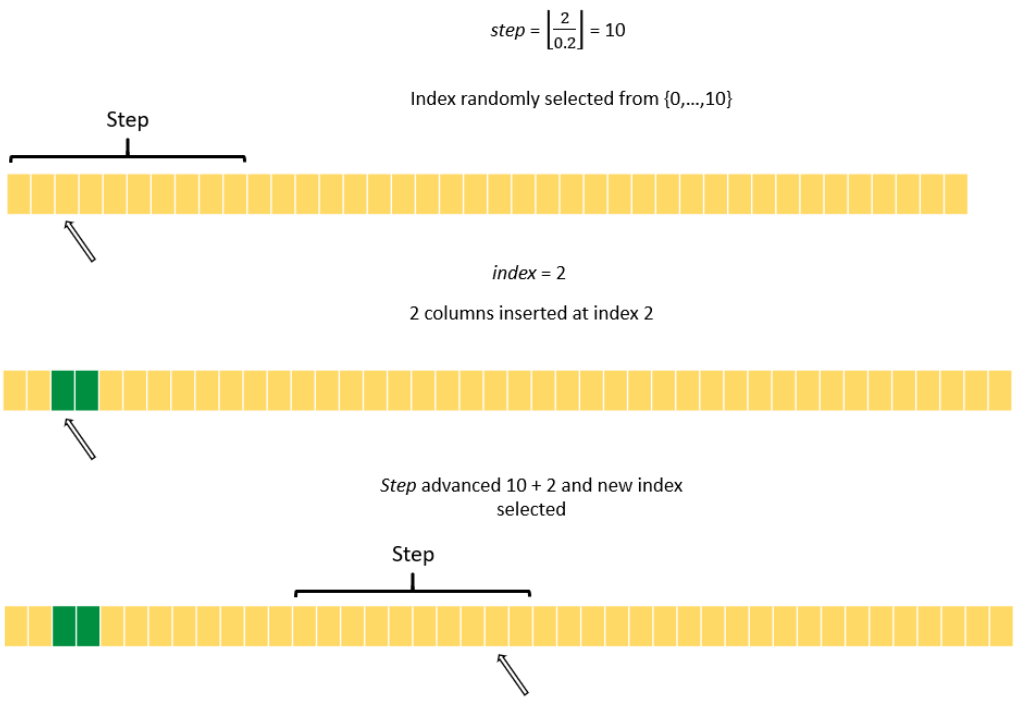

Figure 3.2. Perturbation column insertion

One thing to note is that the floor function in Equation 3.3 causes the resulting BWO to differ slightly from the input saturation s. Table 3.1 details the input settings and resulting BWO values used in our experimentation.

Table 3.1. Bandwidth overhead levels

BWO	**Saturation** (s)	**Width** (w)
5.00%	0.05	5
10.00%	0.1	5
20.00%	0.2	5
31.30%	0.3	5
41.70%	0.4	5
50.00%	0.5	5
62.50%	0.6	5
71.40%	0.7	5
83.30%	0.8	5
100.00%	0.9	5

3.1.1 *Prism* Algorithm

To turn our random perturbations into adversarial examples, we must tune the values using PGD. In order to simulate a real-world scenario, however, we cannot simply pass our entire trace into a PGD function. That would simulate knowledge of the trace before it occurs. We instead use a *perturbation window* to tune the perturbations a little bit at a time. The description of this process is shown in Algorithm 2.

Algorithm 2 Windowed Perturbation Tuning

Input

win_size	Size of perturbation window, s.t. win_size> 0
trace	A trace that has been processed through Algorithm 1
pert_mask	Output from Algorithm 1 associated to trace
pad_mask	Same as in Algorithm 1
μ	The length of the trace

Output

X_batch	A tuned copy of the trace that contains adversarial examples

revealing_mask$[\mu] \leftarrow \{0, 0, ..., 0\}$
protection_mask$[\mu] \leftarrow \{1, 1, ..., 1\}$
window_lower $\leftarrow 0$
window_upper $\leftarrow 0$
X_batch $\leftarrow copy$(trace)
while window_upper $< \mu$ **do**
    i $\leftarrow 0$
    while i < win_size **do**
        revealing_mask[window_upper] $\leftarrow 1$
        window_upper $\leftarrow$ window_upper + 1
        if window_upper = $\mu - 1$ **then**
            break
        end if
        i $\leftarrow$ i + 1
    end while
    X_batch $\leftarrow$ X_batch $*$ revealing_mask
    X_batch $\leftarrow pgd$(X_batch, protection_mask)
    i $\leftarrow 0$
    while i < win_size **do**
        protection_mask[window_lower] $\leftarrow 0$
        window_lower $\leftarrow$ window_lower + 1
        if window_lower = $\mu - 1$ **then**
            break
        end if
        i $\leftarrow$ i + 1
    end while
    X_batch $\leftarrow$ X_batch $*$ revealing_mask + trace $*$ protection_mask
end while

To enforce our windowed approach, we once utilize two masks: one to hide the 'unseen' portion of the trace from the PGD tuning, called *revealing_mask* in the algorithm, and another to protect prior packets in the trace which have already been tuned, called *protection_mask*. To hide the unseen portion of the trace, we begin with an array of 0's equal to the length of our trace. For *protection_mask*, we begin with an array of 1's equal to the length of our trace. In the algorithm, the space between the two masks is our current perturbation window: i.e., packets that have been 'seen' but are not yet protected.

To implement the window, we move the front of the window forward by consecutively setting the values of *window_size* elements in *revealing_mask* to 1 and multiply *X_batch* by *revealing_mask*. This is what 'reveals' the perturbation window. *X_batch* is then passed into the PGD function along with *protection_mask* which is used in PGD to protect the previous portion of the trace from being re-perturbed.

After the packets have been tuned by PGD, we advance the back of the window forward by setting *window_size* elements in *protection_mask* to 0 which will protect the window we just tuned on the next iteration. At this point, we need to reassemble the trace by concatenating the tuned portion with the unseen portion. We accomplish this by taking the sum of *X_batch*revealing_mask* and *trace*protection_mask*. The *X_batch*revealing_mask* multiplication zeros out everything past the current position window in the tuned trace, and the *trace*protection_mask* zeroes out everything before the window in the original trace. By adding these together, we effectively concatenate the two sections of the traces.

3.1.2 PGD Implementation

To perform the tuning on the perturbation window, we use a PGD function that moves the perturbations along the gradient of the loss with respect to the input trace. The loss function we use is the *SparseCategoricalCrossentropy* package from TensorFlow. One can think of the PGD algorithm as a means to minimize the loss of the model by changing the input as opposed to minimizing loss by changing the parameters of the model. By minimizing over the input, we cause the model to misclassify with high confidence. Our PGD algorithm follows in Algorithm 3.

Algorithm 3 PGD

Input

X_batch	As passed from Algorithm 2
protection_mask	As passed from Algorithm 2
pert_mask	Same as in Algorithm 2
pad_mask	Same as in Algorithm 1
model	White box model
y	Correct label for X_batch
steps	The number of steps taken down the gradient
step_size	Size of each step taken down the gradient
ϵ	PGD epsilon value
ρ	Lowest range value seen in original trace
ω	Highest range value seen in original trace
μ	The length of the trace

Output

X_batch	X_batch where the current perturbation window has been tuned

noise[μ] $\leftarrow_{\$}$ Gaussian noise with unit standard deviation
noise $\leftarrow$ noise $*$ protection_mask
X_adv $\leftarrow$ X_batch +.001$*$ noise
i $\leftarrow$ 0
while i < steps **do**
  prediction $\leftarrow$ model.predict(X_adv)
  loss $\leftarrow$ *SparseCategoricalCrossentropy*(y, prediction)
  $\nabla_i \leftarrow$ sgn(∇(loss, X_adv))
  $\nabla_i^* \leftarrow \nabla_i \cdot$ protection_mask $\cdot$ pert_mask $\cdot$ pad_mask
  X_adv $\leftarrow$ X_adv + step_size $\cdot \nabla_i^*$
  X_adv $\leftarrow$ *clip*(X_adv, X_batch - ϵ, X_batch + ϵ)
  X_adv $\leftarrow$ *clip*(X_adv, ρ, ω)
end while

An array the length of the trace is initialized with values from a Gaussian distribution with a mean value of 0 and a standard deviation of 1 which is used to initialize the random

perturbation columns. We then get a prediction classification from the model and get the loss of that prediction. We take the sign of the gradient of the loss and multiply that value by our masks from the input, which ensures that only the perturbation columns in the current perturbation window are changed and that the padding of the trace is not altered. Since our trace is one dimensional, we only need the sign of the gradient to tune the perturbations. At this point, we have an array that only holds gradient values in viable perturbation columns, so we multiply that array by the step size and add it to our tuned copy of the trace. Finally, we clip the values of the tuned trace to lie within our epsilon window as well as between the minimum and maximum values we have seen in the trace.

CHAPTER 4:
Evaluation

In this section, we present our experimental methods and analyze our findings. We first examine the *Prism* defense performance when defending VF traces, followed by our WF results.

For our experimental setup, we trained two Deep Fingerprinting (Section 2.3.1) models on the closed-world dataset, one for each of two *attack settings*: *white box* and *black box*. A white box setting is where the defense has access to the model being used for classification and can create an attack specifically tuned to that model. White box models are only for experimental purposes since it is not realistic to assume the client will have access to the adversary's model. The black box model is a second model trained on the same dataset as the white box model, but the defense algorithm does not see the black box model while tuning the adversarial examples. The black box model realistically models the real-world and is what we use to quantify our resistance to attack. As noted in Section 2.4, adversarial examples are highly transferable, so we expect to see similar results between white box and black box settings. We measure the classification accuracy of the black box model on the adversarial examples generated with the white box model.

4.1 VF Testing

In this section we detail the results of *Prism* when defending against VF models.

4.1.1 The Dataset

Our dataset is modeled closely after that used in [2]. The traces for our dataset that we used to train our VF classifier models were collected using `tcp dump` while visiting websites through a 'Tor crawler' called `tor-browser-crawler` which accesses the web through a Tor client. The dataset is a collection of packet traces for nine Disney video trailers that were streamed over YouTube. These videos ranged in length from 104 seconds to 401 seconds; however, if we exclude the longest video, all other videos fall between 104 and 163 seconds.

The raw data is of the form <Timestamp, ±packet size>, where the timestamp is relative to the beginning of the packet capture and the sign of the packet size value indicates the packet direction—packets outbound from the client are positive, and packets inbound to the client are negative. These traces are closed-world, meaning that the client was only streaming the target video while collecting the trace and was not browsing any other site.

There are approximately 1,300 traffic traces, each representing a different streaming instance for each video, giving us a total dataset size of over 12,500 traces. Differing streaming circumstances may produce different traces for the same video, so having a large number of traces helps to ensure that there is enough data to account for streaming the video under various circumstances.

Preprocessing

From these 1,300 traces, only traces that are longer than 2500 packets are kept. Anything under that is considered a corrupted trace and is discarded. This trimmed the dataset down to approximately 5,200 traces. The traces were then truncated to 15,000 packets, and any traces shorter than 15,000 packets were padded with trailing zeros until the length was 15,000. Truncating the traces speeds up the training process while also improving accuracy by constraining the data.

In other Website Fingerprinting literature, it is standard to scale the traces down to simply [+1, -1] to represent the direction of the packets and ignore packet size [2]–[4], [18]. The literature has found that the majority of the useful information can be extracted by only looking at sequencing, and all other features are extraneous. For Video Fingerprinting, however, we found that our accuracy increased by approximately 7% by retaining the packet size vs. scaling, moving from 85% to 92% when using DF. Due to the significant increase in accuracy, we choose to retain packet sizes throughout our VF experimentation.

4.1.2 Attack Model Training and Hyperparameter Tuning

When training the DF models, we primarily rely on the hyperparameters that are prescribed for WF in [2]. We also reference the work of [7], which adapted DF for use with VF classification. The hyperparameter choices are shown in Table 4.1. The models required more epochs when training on video traces, likely due to the fact that the variance between

given video traces are not as pronounced as variance between various website traces. Additionally, we found that early stopping was necessary as some models would begin to overfit, and others continued to improve in performance through all 90 epochs.

Table 4.1. Deep fingerprinting hyperparameters

Hyperparameters	**Video Fingerprinting**	**Website Fingerprinting (Defaults [2])**
Training Epoch	90	30
Batch Size	128	128
Optimizer	Adamax	Adamax
Learning Rate	.002	.002
Early Stopping Enabled	Yes	No

4.1.3 Defense Tool Hyperparameter Tuning

When we began testing the *Prism* defense, we started with the *initial* hyperparameters detailed in Table 4.2. For each of the hyperparameters, our decisions for the starting points were:

- **Perturbation Window Size**: We wanted to pick a window size that is realistic to implement in the real-world. Maintaining a packet queue of 50 is large enough to form an efficient defense over the trace, while being small enough that it will not significantly impact network latency.
- **PGD Steps**: This is the number of steps of size *PGD Step Size* that the PGD attack takes down the gradient. We used 100 as a starting point because we wanted a value that gave enough granularity to find minimums but that did not take too long to tune.
- **PGD Step Size**: The step size indicates the value jump for each step when descending the gradient in PGD. In conjunction with the number of steps at 100, this gives a value range of 10,000 from the starting point.
- **Epsilon**: The epsilon value is used to limit the range the perturbation values are allowed to differ from the original random perturbations, used at the end of the protocol. It was originally set at 4,500 as that is toward the upper range of the positive range we see in the data set.

The results of the defense with the initial hyperparameter settings was lackluster, and we had a difficult time getting either white box or black box settings to reduce the model accuracy below 70%.

Table 4.2. VF defense hyperparameters

VF Prism Hyperparameters	**Initial**	**Tuned**
Perturbation Window Size	50	50
PGD Steps	100	40
PGD Step Size	100	1500
PGD Epsilon	4500	60000

To tune the hyperparameters, we tested various hyperparameters against a DF model trained on a 20% BWO dataset. We first began by increasing epsilon to 60,000. Our initial thought with setting epsilon at 4,500 was to limit the change in value to the positive max range, but that assumption did not account for the much larger negative max range (up to -60,000) indicating large incoming packets in the dataset. To account for the larger epsilon, we needed to add an additional step that clipped our perturbations to lie between -60,000 and 5,000. By only increasing epsilon, we were able to reduce model accuracy for white box to 54%.

The purpose of epsilon is to provide a range of allowable variation from the original values of the trace. By increasing epsilon to 60,000, we allow the value of the adversarial example to differ by as much as 100% from the original value. Since we are modeling a bounded network system, however, we clip the values of the adversarial examples so they lie within the range of observed network traffic. This ensures that the models are not fooled simply by impossible transmission sizes present in the trace.

This brings up an important discussion on what defines “noise” in our experimentation. In a traditional adversarial example scenario where the goal is to misclassify images, the measurement of noise refers to the distance metrics which quantify how much individual pixels in the image change as well as how much the overall image changed. The reason being that when attacking the image, the goal is to keep the image close to the original as perceived by humans. In a network scenario, however, we do not have the liberty to make

small adjustments to packets across the entire trace, as that would break the networking protocol, nor do we have the liberty to add an unlimited amount of perturbation packets. Every added perturbation packet incurs a cost of adding bandwidth overhead to an already bandwidth starved system such as Tor. Instead, we measure noise as the number of additional packets that must be added to the trace to fool the classifier (BWO). In this sense, we only care about how much the trace *as a whole* differs from the original, not how the packet individually differs.

The performance improvement resulting from the increase in epsilon indicated that the PGD algorithm needed more freedom to adjust the values of the perturbations, so our next step was to increase the step size to increase the possible range of variance from the original value. Our range was originally 10,000 (steps * step size), so we increased the step size to 500 (50,000 range) and tested again. With these two changes, our approach now defeated both white box and black box settings, with a white box accuracy of 1% and a black box accuracy of 34%.

At this point, we were able to defeat the classifier, but the adversarial example generation took about 50 seconds per trace (15,000 packets) which was slower than we wanted. To speed up the program, we decreased the number of PGD steps to 40 and increased the step size to 1,500. This gave us a max range of 60,000. By decreasing the number of steps, we were able to significantly speed up the defense while still defeating the model, with the classifier reporting almost 0% accuracy. Each trace now takes roughly 18 seconds to tune which should not be noticeable when evenly distributed over a 3 minute video download. In our testing, we do not simulate the delay while waiting for the perturbation window to fill, so our latency calculations are a lower bound on the time required to perturb the trace. Future work will be needed to determine latency in a real-world setting.

4.1.4 VF Defense Results Analysis

One of the goals throughout the experimentation process was to determine the lowest BWO where the trace defense still succeeded: i.e., where both white box and black box testing resulted in model accuracy <10%. Thus, after determining the *Prism* hyperparameters that yield the best results, we tested the approach over various BWO levels, ranging from 5% to 100% BWO. The exact BWO values tested are shown in Table 3.1.

Our experimentation revealed that it was important to have a different pair of models trained for each BWO level we tested. We found that if a model was trained at a given BWO, changing the BWO of a trace by as little as 10% would cause the model accuracy to decrease by approximately 75% when tested on random perturbations, essentially defeating the classification. To accurately test *Prism*, we trained a white box and black box model for every BWO level tested.

For all of our testing, we used a test sample size of 125 traces. We were limited in the number of traces we could test at one time due to computational capabilities: too many traces would overload GPU memory. Since the traces of the test set are randomly sampled from the dataset, we are able to observe the performance on said traces and generalize the results to the entire dataset.

Our initial testing modeled perturbation packets being inserted in the trace from both client and Tor guard. In our implementation, this meant that we allowed the perturbation values to to range from -60,000 to 5,000, where positive values indicate dummy packets originating from the client. The results of this test are shown in Figure 4.1.

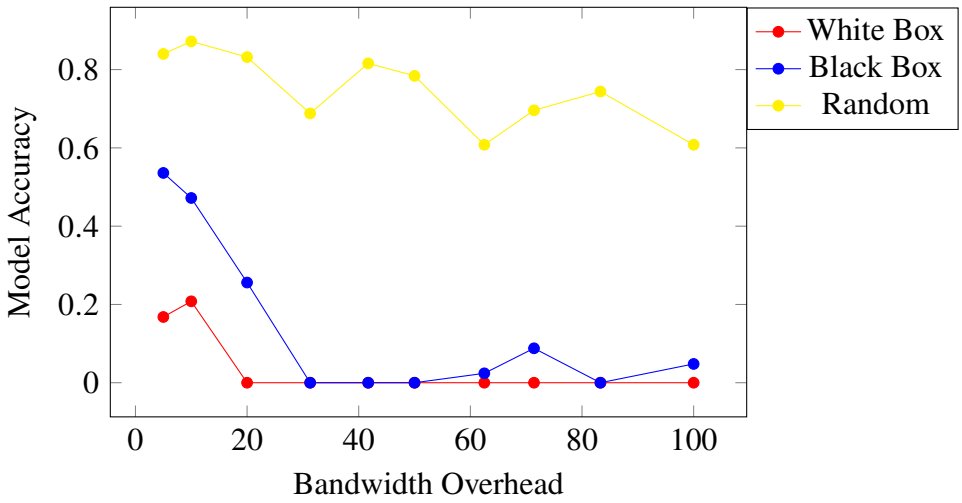

Figure 4.1. Results of Video Fingerprinting Server-Client *Prism* defense on an undefended DF model

The trend in the graph indicates that defense performance increases as BWO increases until 30% BWO. At this point, both models are completely defeated, classifying with 0%

accuracy, and increasing BWO cannot improve defense performance. The accuracy of the model on random noise decreases as BWO increases, but remains fairly consistent, differing by 25% across the various BWO ranges and demonstrating that random dummy packets will not significantly affect the classification model.

While the results in Figure 4.1 are promising, the defense relies on the client having access to a GPU to compute the perturbation on the fly as the video is being downloaded. Not all users will have access to a GPU. As such, it was necessary to test if our defense was still effective when only the Tor guard sent perturbation packets.

To model this scenario in our test, we created a new dataset from the original, but when inserting the columns of random perturbations, we restricted the range to only negative values between -60,000 and -100. We then trained new models for every BWO level. By limiting the range of random perturbations, we ensure that the defense does not have an unfair advantage: i.e., the model does not expect to see a disproportionate number of outbound packets. In our defense model, we kept everything the same as before except we now clipped the adversarial examples so they ranged from -60,000 to -100. The results of the test are shown in Figure 4.2

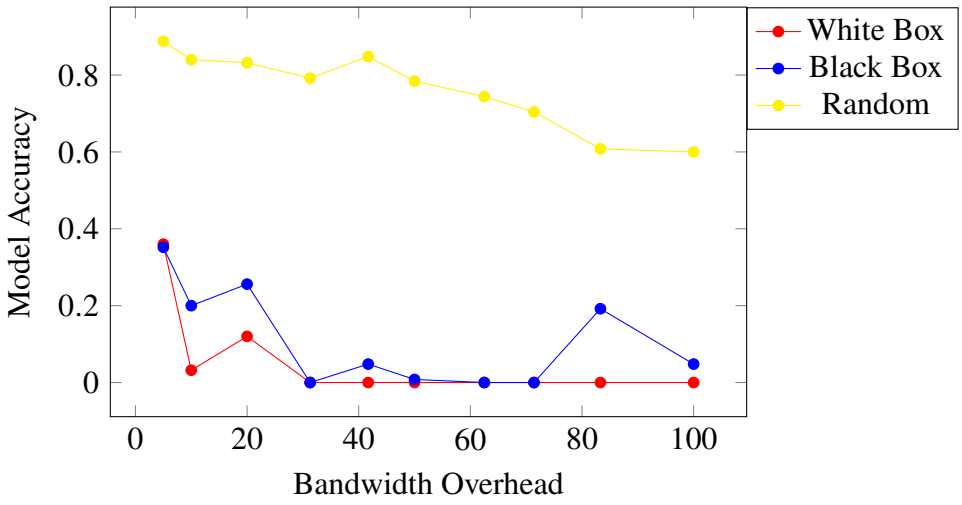

Figure 4.2. Results of Video Fingerprinting server side only *Prism* defense on an undefended DF model

The results of this test follow similar trends as observed in the server-client defense. The

defense performance increases as BWO increases up to 30%, decreasing model accuracy from 38% down to 0%, and the accuracy on the random perturbations decreases by 25% across the BWO values. These results are very promising, as they indicate that the defense may be implemented at the Tor guard nodes, increasing the availability of the defense to users who may not have access to GPUs.

At this point, it is important to discuss our decision to use unscaled traces in our dataset. When scaled, the server side only defense will be represented as bursts of [-1] for the perturbation column width. Our testing did not include how this affects the performance of the defense, especially on a defended model. However, as previously discussed, should the VF attacker choose to scale the data, he will be incurring a significant accuracy reduction with the model. Testing the defense in these various scenarios remains a topic for future work.

To test *Prism* performance in a more realistic real-world setting, we included adversarial examples in the training set at 30% BWO. We choose 30% BWO because that is lowest BWO level that *Prism* defeats both the white box and black box models. We found that when we trained the model entirely on adversarial examples, the resulting model was less robust and misclassified any trace that did not contain adversarial examples that were generated on an undefended model, even random noise. To correct this, we split the training set so that half of the traces contained random perturbations and half contained adversarial examples. We then took this defended model, created another set of adversarial examples for the training dataset, and created another defended model using the adversarial examples from the second step. This way, the adversarial examples that our final defended model was trained on were crafted on a defended model to increase the likelihood that the model was resistant to adversarial examples. The results of this testing are summarized in Table 4.3.

Table 4.3. Defended VF model results

Model Accuracy With:	**White Box**	**Black Box**	**Random**
Server-Client–only adversarial training	0%	0%	15.2%
Server-Client–split adversarial training	0%	0.5%	76%
Server Only–only adversarial training	0%	0%	4%
Server Only–split adversarial training	0%	0%	74%

The *Prism* defense defeats the attacker's model even with adversarial training. For both black-box and white-box, we observed ≈0% accuracy for detecting the user's video. The results show how models that were only trained on adversarial examples were not effective at all, even misclassifying random noise. When we include random perturbations back into the test set along with the adversarial examples, we see that the model correctly classifies benign traces with higher accuracy, but the adversarial training does not increase performance on defended traces.

4.2 WF Testing

In this section we detail the results of *Prism* when defending against WF models.

4.2.1 The Dataset

For our WF testing, we used the dataset collected by Sirinam et al. [2] (accessed via Google Drive). A detailed description of this dataset can be found in [2]. As stated in Section 4.1.1, the dataset used for our VF experimentation is very similar to that used in Deep Fingerprinting, with the most noticeable difference being that our dataset for VF is *not* scaled to [+1, -1] to only represent traffic direction. For a detailed description on why we do not scale our VF dataset, refer to Section 4.1.1. Since the dataset only included values of -1 and 1, we had to ensure that when inserting the random perturbation columns we initialized the values randomly to either -1 or 1 as well.

4.2.2 Attack Model Training

To train our DF models for WF, we rely on the default hyperparameter settings as prescribed in [2]. These are shown in Table 4.1.

4.2.3 Defense Tool Hyperparameter Tuning

Following the logic of our hyperparameter tuning described in Section 4.1.3, we choose hyperparameter values that gave the defense the freedom to shift the adversarial example values to anything seen in the range of the trace: e.g., the ability to shift a '1' to a '-1'. We found that a larger step size significantly improved the performance of the defense. We hypothesize that this allowed the defense to traverse far down the gradient and increased the chance that the value was flipped. We also eliminated the threshold on perturbation size by removing epsilon from the equation because we did not want to restrict the ability of the defense to flip values. Our resulting hyperparameter settings are summarized in Table 4.4.

Table 4.4. WF defense hyperparameter settings

WF Prism Hyperparameters	**Value**
Perturbation Window Size	50
PGD Steps	12
PGD Step Size	12
PGD Epsilon	∞

4.2.4 WF Defense Results Analysis

Similar to our VF testing (Section 4.1.4), one of our goals with our WF testing was to determine the lowest BWO level where the defense succeeds. We accomplish this by testing our defense against a WF classifier at all BWO levels listed in Table 3.1. We also train models at each BWO level for the reasons described in Section 4.1.4.

As with our VF testing, we test our defense on 125 traces. Our test models both client and server participation in the defense. In our implementation, this meant that we allowed the adversarial example values to be either -1 or 1. We exclude 0 as that would represent an impossible non-transmission. The results of this test are shown in Figure 4.3.

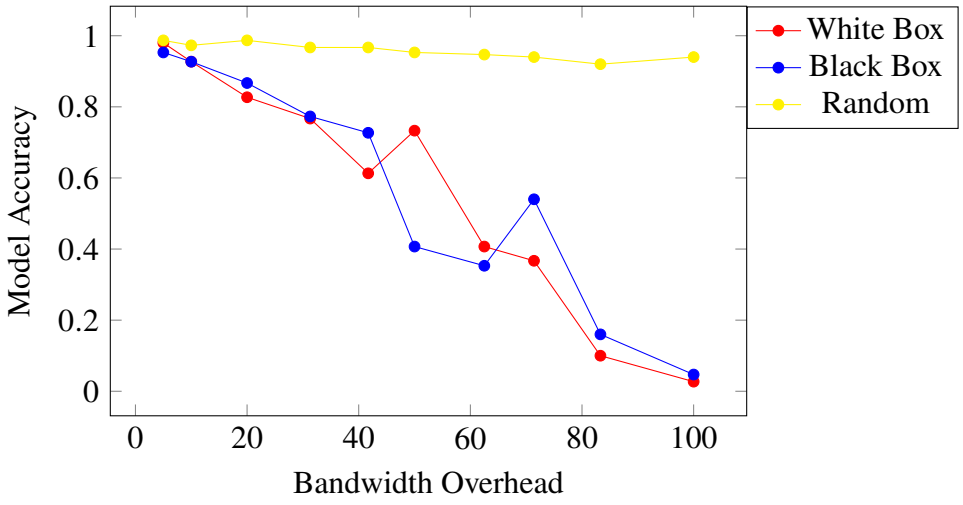

Figure 4.3. Results of Website Fingerprinting Server-Client *Prism* defense on an undefended DF model

These results indicate that the *Prism* defense succeeds when defending website traces, reducing model accuracy from 97% down to 0% at 100% BWO. One thing to note is that a much higher BWO level is needed to defeat the attack in this scenario. This could be due to the fact that the traces are scaled, or because the model performs better when classifying website traces rather than video traces. Future work can explore the cause for robustness in the website fingerprinting DF model, and attempt to build better VF classifiers as well as a better defense against WF.

To test our model performance in a more realistic real-world setting, we included adversarial examples in the training set at 60% BWO. Similar to our VF defended testing, we include results on a model trained only on adversarial examples generated from an undefended model and a model trained half on random perturbations and half on adversarial examples generated from a defended model. The results are shown in Table 4.5.

Table 4.5. Defended WF model results

Model Accuracy On:	**White Box**	**Black Box**	**Random**
Server-Client–only adversarial training	0.8%	1.0%	3.5%
Server-Client–split adversarial training	26.9%	55.6%	91.1%

Prism defense reduces the accuracy of the attacker's model to 55.6%. While this does not completely defeat the model, it reduces certainty to approximately a 50/50 chance, or in other words, it reduces the likelihood to a coin flip as to whether the prediction is correct for the adversary. Similar to VF, we see that the split training set significantly increases accuracy on benign traces. While adversarial training increases accuracy on the black box model by $\approx$ 13%, it does not overcome the defense.

CHAPTER 5:
Discussion

The results of our study show that adversarial examples are a viable option for defending Tor against various fingerprinting attacks. In particular, we show that *Prism* can effectively defeat VF attacks and significantly reduce the certainty of WF attacks in a simulated real-world environment, even defeating DF, a state of the art DNN that is able to overcome most defenses available at the time of this publication [2]. In this section, we discuss possible design decisions for implementation and limitations of the defense as well as possible future work.

5.1 Possible Improvements

With *Prism*, we propose a novel method of generating adversarial examples for situations where the data being perturbed is discovered over time. We show that our approach is effective at defeating both image classification and fingerprinting classification, but more research is needed to optimize this approach.

5.1.1 Decaying the Step Size

One possible improvement for *Prism* would be adding a decay to the step size in PGD. Since our goal with scaled data is to induce a flipped sign when it minimizes loss, adding a bias to earlier gradients may yield better results, as the initial gradients are more representative of the loss in the model. Adding this decay could also reduce the number of steps required when crafting adversarial examples, which would increase the speed at which adversarial examples can be generated.

5.1.2 Server-Client Participation

We include testing on a network scenario where both the server (guard node) and the client participate in the defense, enabling dummy packet transmissions in both directions. In this scenario, however, it is likely that the adversarial example packets computed at the server would be different than what is calculated at the client, which could potentially result in a

reduction in performance of the defense. While we show that the defense can feasibly be implemented only at the server when defending VF traces, participation from both parties is crucial when the dataset is scaled by the adversary, as discussed in Section 4.1.

A possible solution would be to compute the adversarial examples entirely at one side: i.e., the server or the client, and include information about how the opposite party is to participate in the dummy packet burst in the first transmitted packet. This would require that the first packet in each dummy packet burst originate from either the server or the client, thereby reducing the sample space of possible adversarial example sequences. Our testing does not include how this might effect the performance of the defense but could be explored in future work.

5.2 Adversarial Training

Our testing revealed that adversarial training does not significantly improve model performance when classifying traces defended with *Prism*. There are other methods of defending against adversarial examples as discussed in [19] and [16]. Future work can research other defenses to determine whether *Prism* remains effective against a variety of defended models, however, as [16] points out, defenses against adversarial examples do not always improve model performance.

5.3 BWO Requirements

In our testing we include results from various BWO levels on undefended models. This testing acts more as a proof of concept of our defense, however, because in a real-world setting, the attacker would train on defended traces observed "in the wild." We only include a single BWO level for our defended models due to time limitations while performing the research. As discussed in Section 2.1, Tor is a bandwidth-starved environment, so minimizing BWO is crucial in a defense. Exploring a range of BWO performance on defended models would give an indication of the feasibility of implementing the *Prism* defense in Tor.

5.4 Open-World Testing

In this paper, we only explore the defense in a closed-world setting. Closed-world testing gives an advantage to the classifier since there is no noise present when classifying the traces. Future work will be needed to test *Prism* in an open-world setting where we predict that it will yeild better results.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusion

In this study, we explored the performance of a defense for both VF and WF attacks. We proposed *Prism*, a defense that creates adversarial examples using a perturbation window and an adaptation of projected gradient descent (PGD) to create perturbations that are compatible with a real-time networking environment. We analyzed the *Prism* defense across BWO levels varying from 5% to 100% on both VF and WF Deep Fingerprinting (DF) models, and show that for both scenarios, *Prism* defeats the fingerprinting attacks by reducing model accuracy to 0% at varying BWO levels. Finally, we demonstrated that *Prism* completely defeats defended VF models by reducing accuracy to 0%, and significantly reduces defended WF model performance by reducing accuracy to 55.6%.

Our study demonstrates that real-world defenses against fingerprinting attacks that do not incur heavy BWO costs are possible and that more research is needed in this field to develop an effective defense on Tor.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, Tech. Rep., 2004.

[2] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1928–1943.

[3] M. S. Rahman, M. Imani, N. Mathews, and M. Wright, "Mockingbird: Defending against deep-learning-based website fingerprinting attacks with adversarial traces," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1594–1609, 2020.

[4] T. Wang and I. Goldberg, "Walkie-talkie: An efficient defense against passive website fingerprinting attacks," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1375–1390.

[5] S. Shan, A. N. Bhagoji, H. Zheng, and B. Y. Zhao, "A real-time defense against website fingerprinting attacks," 2021. Available: https://arxiv.org/abs/2102.04291

[6] A. Barton, M. Wright, J. Ming, and M. Imani, "Towards predicting efficient and anonymous tor circuits," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 429–444.

[7] D. Campuzano, Carlos, "Towards video fingerprinting attacks over tor," 2021. Available: https://calhoun.nps.edu/handle/10945/68304

[8] A. Barton and M. Wright, "Denasa: Destination-naive as-awareness in anonymous communications." *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 4, pp. 356–372, 2016.

[9] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of tor," in *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 2005, pp. 183–195.

[10] M. Imani, A. Barton, and M. Wright, "Guard sets in tor using as relationships." *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 1, pp. 145–165, 2018.

[11] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[12] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting." *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 4, pp. 21–36, 2016.

[13] 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*, 09 2016, vol. 9878, pp. 27–46.

[14] M. Shen, Z. Gao, L. Zhu, and K. Xu, "Efficient fine-grained website fingerprinting via encrypted traffic analysis with deep learning," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10.

[15] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *2nd International Conference on Learning Representations, ICLR 2014*, Jan. 2014, 2nd International Conference on Learning Representations, ICLR 2014 ; Conference date: 14-04-2014 Through 16-04-2014.

[16] A. Barton *et al.*, "Defending neural networks against adversarial examples," Ph.D. dissertation, University of Texas Arlington, 2018.

[17] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[18] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer, "Adversarial patch," 2017. Available: https://arxiv.org/abs/1712.09665

[19] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[20] J. Lu, T. Issaranon, and D. Forsyth, "Safetynet: Detecting and rejecting adversarial examples robustly," 2017. Available: https://arxiv.org/abs/1704.00103

[21] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," 2014. Available: https://arxiv.org/abs/1412.5068

[22] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014. Available: https://arxiv.org/abs/1412.6572

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California