Cache Shaping: An Effective Defense Against Cache-Based Website Fingerprinting

Haipeng Li
University of Cincinnati
Cincinnati, OH, USA
li2hp@mail.uc.edu

Nan Niu
University of Cincinnati
Cincinnati, OH, USA
nan.niu@uc.edu

Boyang Wang
University of Cincinnati
Cincinnati, OH, USA
boyang.wang@uc.edu

ABSTRACT

Cache-based website fingerprinting attacks can infer which website a user visits by measuring CPU cache activities. Studies have shown that an attacker can achieve high accuracy with a low sampling rate by monitoring cache occupancy of the entire Last Level Cache. Although a defense has been proposed, it was not effective when an attacker adapts and retrains a classifier with defended data. In this paper, we propose a new defense, referred to as *cache shaping*, to preserve user privacy against cache-based website fingerprinting attacks. Our proposed defense produces dummy cache activities by introducing dummy I/O operations and implementing with multiple processes, which hides fingerprints when a user visits websites. Our experimental results over large-scale datasets collected from multiple web browsers and operating systems show that our defense remains effective even if an attacker retrains a classifier with defended cache traces. We demonstrate the efficacy of our defense in the closed-world setting and the open-world setting by leveraging deep neural networks as classifiers.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; **Network security**.

KEYWORDS

Website fingerprinting; machine learning; CPU cache; defense

ACM Reference Format:
Haipeng Li, Nan Niu, and Boyang Wang. 2022. Cache Shaping: An Effective Defense Against Cache-Based Website Fingerprinting. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3508398.3511500

1 INTRODUCTION

Website fingerprinting attacks can infer which website a user visits by eavesdropping encrypted network traffic [12, 14, 15, 19, 27, 28, 31, 36, 40] or monitoring cache activities [6, 25, 35]. Revealing which websites that a user visits harms user privacy and may lead to other sensitive information leakage, such as identities, locations, etc. While effective defenses [12, 13, 16, 36, 41] have been proposed against traffic-based website fingerprinting, how to design effective defenses against cache-based website fingerprinting remains open.

Specifically, a recent study [35] by Shusterman et al. shows that an attacker can achieve more than 87% accuracy in cache-based website fingerprinting by monitoring cache occupancy of the Last Level Cache on a user's machine and leveraging a Convolutional Neural Network as the classifier. Although a defense (named *cache masking*) was proposed in [35], the defense is not effective when an attacker adapts and retrains the classifier with defended data. For instance, an attacker can still achieve 73% accuracy when it retrains the classifier with defended data generated by cache masking. As this cache-based website fingerprinting attack monitors the cache occupancy of the entire Last-Level Cache rather than specific cache sets, other countermeasures [11, 20, 21, 29, 42] against traditional cache-based attacks are also ineffective.

In this paper, we proposed a new defense, referred to as *cache shaping*, to effectively defend against cache-based website fingerprinting. Specifically, an attacker will derive low accuracy in cache-based website fingerprinting even if it adapts to our defense and retrains a classifier with defended data. The core idea of cache shaping is to introduce dummy cache activities when a user loads a website in a web browser, such that the cache patterns of different websites are difficult to distinguish. The main contributions and findings of this study are summarized as below:

- We build a new defense against cache-based website fingerprinting attacks. We first examine why the previous defense is not effective, and then design dummy cache activities in our defense to hide fingerprints by (1) simulating dummy rendering process of a web browser with dummy I/O operations and (2) implementing with multiple processes.
- We collect real-world large-scale datasets (16 GBs with more than 270,000 cache traces) on three common web browsers (Chrome, Firefox, and Tor Browser) and two popular operating systems (Linux and Windows). We evaluate the efficacy of our defense in the closed-world setting and the open-world setting by leveraging deep neural networks (including Convolutional Neural Networks and Long Short-Term Memory) as classifiers.
- As a necessary trade-off, our defense causes higher CPU slowdowns than the previous defense [35]. Specifically, according to our experiments, cache masking introduces only 20% CPU slowdowns on integer benchmarks and 24% CPU slowdowns on floating point benchmarks while our defense can introduce 57% CPU slowdowns on integer benchmarks and 52% CPU slowdowns on floating point benchmarks.
- To mitigate the CPU slowdowns caused by our defense, we also propose a method, named *random switch*, which can be

integrated with our defense. Random switch can randomly turn on and off perturbations (i.e., dummy operations) introduced by our defense. For instance, it can reduce the CPU slowdowns on integer benchmarks and floating point benchmarks to 49% and 43% respectively.

- We examine how to proactively *detect* cache-based website fingerprinting as an additional protection to minimize overheads. Our results show that, by monitoring the cache activities of a website for only 1 second, our detection can identify malicious websites running cache-based website fingerprinting with 98.5% precision and 83.2% recall.

Reproducibility. The source code and datasets of this study are publicly available at [1].

2 SYSTEM AND ATTACK MODEL

System and Attack Model. Our system in Fig. 1 includes two parties, a user and an attacker. This user visits a sensitive website using a web browser on a computer with or without using anonymous networks, such as VPN and Tor. The attacker is *remote*, i.e., the attacker is not able to eavesdrop the user's network traffic, but aims to infer which website the user visits by collecting cache activities on the user's computer. The collection of cache activities can be done through malicious JavaScript code written by the attacker. This type of attacks is referred to as *cache-based website fingerprinting.*

As mentioned in [35], the malicious JavaScript code is hosted on a malicious website (either directly controlled by the attacker or injected by the attacker without being detected). In addition, the attacker assumes that the user opens two tabs in a web browser. Specifically, the user opens the first tab to visit the malicious website (e.g., through links from phishing emails) and opens the second tab to visit a sensitive website while the first session remains on.

A sequence of cache activities on the user's computer visiting one sensitive website is denoted as a *cache trace.* A cache trace, which is a 1-dimensional time-series data, can be denoted as $\vec{v} = (v_1, ..., v_n)$, where v_i is the measurement of the corresponding cache activity on the user's computer at sample point i. Each measurement/sample can be measured in terms of access delay, which is proportional to the amount of data processed by the Last-Level Cache of the user's computer [25, 35]. For instance, a longer access delay indicates more data are processed. We assume that the sensitive website is loaded on the user's computer starting from sample point $i = 1$, and n is the total number of samples in a cache trace.

A website fingerprinting can be formulated as a supervised learning problem. Specifically, we assume that an attacker can collect training cache traces with known labels (i.e., websites) on his own machine to train a classifier. Then, the attacker obtains unlabeled test cache traces from the user and predicts the labels of these test traces. By following the previous study in [35], we assume that the training traces and test traces are collected from the same setting, including the same operating system (OS), same web browser, and same last-level-cache size.

Closed-World Setting and Open-World Setting. A website fingerprinting attack can be evaluated in the closed-world setting and the open-world setting. In the closed-world setting, an attacker knows a set of monitored websites, and each cache trace from a user

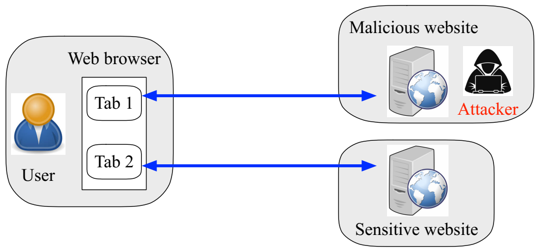


Figure 1: The system and attack model of cache-based website fingerprinting.

corresponds to one of the monitored websites. Given a trace, an attacker infers which monitored website it belongs to. In essence, the closed-world evaluation is a multi-class classification. We leverage accuracy to examine the performance of closed-world evaluation.

In the open-world setting, an attacker knows a set of monitored websites, but a user can visit unmonitored websites in additional to monitored websites. An attacker infers whether a trace is associated with a monitored website or an unmonitored website. The open-world evaluation is a binary classification, which can be evaluated with precision, recall, and precision-recall curve.

In the open-world evaluation, we follow the *standard model* defined in previous work [36, 37]. Specifically, all the traces from unmonitored websites are considered as a single class. This single class will be added to the classifier obtained from the closed-world evaluation as one additional class. This classifier will be re-trained with traces from monitored websites and unmonitored websites. Given an unlabeled trace, if the highest confidence of this classifier belongs to one of the monitored websites and this confidence is greater than a threshold, this trace is considered as a trace associated with a monitored website. Otherwise, it is considered as an unmonitored website. This threshold can be tuned in experiments to seek trade-offs between precision and recall.

3 BACKGROUND

3.1 Cache Architecture

Due to the high access latency between a CPU and RAM (Random Access Memory), *cache memory* (or *cache* in short) was introduced to improve data access time on modern computers. For instance, a cache can store data that were recently accessed by the CPU. If the CPU needs to access data from the same memory address again, fetching the data from the cache renders faster access than retrieving the data from RAM. A typical cache hierarchy in a modern CPU consists of 3 levels, including Level 1 (L1), Level 2 (L2) and Level 3 (L3). Level 3 cache is often referred to as the Last-Level Cache (LLC). Each CPU core has its own L1 cache and L2 cache. The Last-Level-Cache is shared across all the CPU cores.

The LLC is typically organized in a set-associative structure. Specifically, the LLC is divided into cache sets and each cache set includes multiple cache lines. All the memory addresses in RAM are divided into subgroups, where each subgroup of memory addresses will map to one corresponding cache set. Within a cache set, each cache line can store data from any of the memory addresses from the corresponding subgroup. When the CPU needs to fetch data, it first examines whether the address of the data is present in the

LLC. A *cache hit* indicates the requested memory address is in the cache and results in a shorter access time. A *cache miss* suggests the requested memory address is not in the cache, and the CPU needs to fetch the data from the RAM, which causes a longer access time. In addition, if it is a cache miss, some data previously in the cache need to be evicted to make space for the fetched data. How data are evicted is out of the scope of this paper.

3.2 Cache-Based Website Fingerprinting

As the LLC is shared by multiple cores, it creates unintended *side channels* between programs executed on the CPU. For example, a program executed on one core can monitor the memory access activities of a program on a different core through cache hits and cache misses on the LLC. These side channels can lead to privacy leakage, such as cryptographic keys [22, 26] and keystrokes [17], as shown in previous studies. Attacks leveraging these side channels are often referred to as *cache-based attacks*. Cache-based attacks can be launched *locally* (e.g., running a malicious program on a target computer) or *remotely* (e.g., measuring cache activities using JavaScript code through web browsers) [25, 35].

Prime+Probe. Prime+Probe [26] is one of the primary techniques to carry out cache-based attacks. It measures the cache activities with a sequence of iterations. Within each iteration, Prime+Probe consists of 3 steps. In Step 1, the attacker *primes* the cache by filling some cache sets with his own data. The content of his data does not matter. In Step 2, the attacker waits some time to allow the target program to execute. In Step 3, the attacker probes the cache by measuring the access time to the cache sets it primed in Step 1.

If the victim program in Step 2 accessed memory addresses that map to a cache set primed by the attacker in Step 1, the victim program evicted the attacker's data in the cache set. This leads to a cache miss and a longer access time when the attacker probes data in the cache set in Step 3. On the other hand, if the victim program did not access memory addresses that map to a cache set primed in Step 1, it results a cache hit when the attacker probes in Step 3.

The key for a cache-based attack is to be able to distinguish cache hits and cache misses, and thus distinguish which data were used by the victim program. As a result, a cache-based attack using Prime-Probe often requires a high resolution timer. Previous study [25] was able to perform website fingerprinting with Prime+Probe at a small scale (e.g., 10 websites in the closed-world setting). To prevent cache-based attacks from running remotely through web browsers, major web browsers, such as Chrome and Firefox, have disabled features with sub-microsecond resolution timers, which prevents attackers from distinguishing cache hits and cache misses.

A Recent Attack Using Cache Occupancy Channel. As attacks using Prime+Probe are no longer effective due to the lower resolution timers in the latest version of web browsers, Shusterman et. al. [34, 35] proposed a new cache-based website fingerprinting attack by measuring the *cache occupancy channel* through JavaScript code. Specifically, an attacker measures the cache activities of the entire LLC rather than specific cache sets. In Step 1, the attack allocates an LLC-size buffer and fetches this buffer to the LLC. In Step 2, the attacker waits for some time to allow the victim program to execute. In Step 3, the attacker examines the access time to the LLC-size buffer by fetching the LLC-size buffer to LLC.

As the buffer occupies the entire LLC in Step 1, if the victim program accesses data in Step 2, it inevitably evicts some contents of the buffer in order to store the data in cache. As a result, the evicted content causes cache misses and therefore longer delays when the attacker accesses the buffer again in Step 3. According to [35], *the time to access the LLC-size buffer is approximately proportional to the number of cache lines that the victim program uses* in Step 2.

In other words, this attack can measure the amount of data usage of the victim program. The attacker can periodically repeat Step 2 and Step 3 to record the amount of data usage of the victim program for a series of time. As this attack does not rely on distinguishing a cache miss and a cache hit on a specific cache line, it does not need a high resolution timer as previous cache-based attacks and can be carried out with the latest version of web browsers. For instance, with a sampling rate of 500Hz in Chrome, Shusterman et. al. [35] shown that an attacker can achieve more than 87% accuracy in a closed-world setting of 100 websites.

3.3 Limitations of the Existing Defense

Cache Masking. To mitigate the leakage in the cache-based website fingerprinting attack, Shusterman et. al. [35] proposed a defense named *cache masking*. The main idea is to perturb the web browser's original cache activities on the LLC by introducing dummy cache activities. Specifically, the proposed defense method allocates a LLC-size buffer and accesses every cache line in a loop, which repeatedly evicts the entire LLC to hide the original cache activities of a web browser. The authors implemented a proof-of-concept of this defense with `Mastik` side-channel toolkit [43].

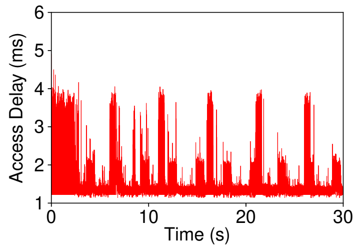
Unfortunately, this defense is not able to effectively lower attack accuracy when an attack adapts and retrains neural networks with defended cache traces. For instance, the attack accuracy only drops from 79% to 73% on Firefox over 100 websites in the closed-world setting as reported in [35]. In other words, cache masking is not effective against real-world attackers who can adapt to the defense.

Why Is Cache Masking Ineffective? To understand why cache masking is ineffective, we first collect cache traces of two websites (`oracle.com` and `wikipedia.com`) when there is no defense and then collect cache traces of these two websites when cache masking is on. We use the malicious JavaScript code and the source code of cache masking from [34, 35] to capture the traces and carry out the defense. The original cache traces and defended cache traces (produced by cache masking) is illustrated in Fig. 2.

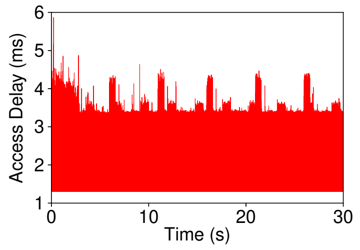
First, we observe that a defended cache trace is very different from the original cache trace. This explains the low accuracy when an attacker trains with non-defended cache traces and tests with defended traces reported in [35]. On the other hand, peaks in cache traces can still be observed in the defended ones of the same website. This explains the high accuracy when an attacker trains and tests with defended cache traces reported in [35].

4 CACHE SHAPING: AN EFFECTIVE DEFENSE

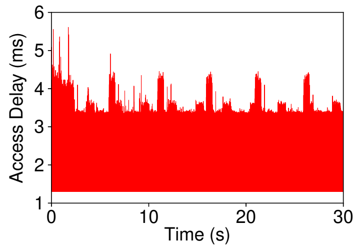
In this section, we introduce our defense, referred to as cache shaping, which can effectively defend against cache-based website fingerprinting. The main idea of our proposed defense is to introduce dummy cache activities that are high enough to hide peaks in original cache traces of different websites. In other words, cache traces



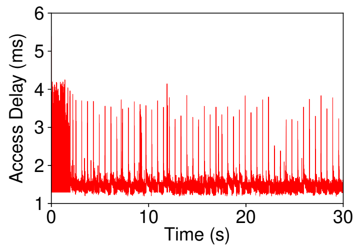
(a) Original trace A (oracle.com)



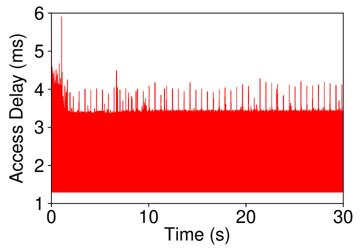
(b) Defended trace B (oracle.com)



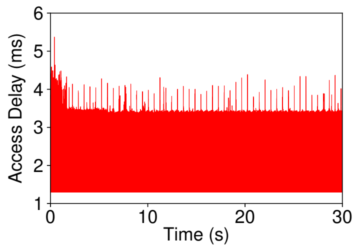
(c) Defended trace C (oracle.com)



(d) Original trace D (wikipedia.com)



(e) Defended trace E (wikipedia.com)



(f) Defended trace F (wikipedia.com)

Figure 2: Cache traces collected on a Linux machine with Chrome. Defended traces are generated by cache masking [35].

Algorithm 1: Cache Shaping in One Process

```
Input: $n$ dummy files $\{f_1, .., f_n\}$, all the $n$ memory pages
    $\{mp_1, ..., mp_n\}$ in LLC, and counter $c = T$, where $T > 0$;
while $c > 0$ do
    for memory page $mp_i$ in LLC do
        for each cache line in memory page $mp_i$ do
            read cache-line-size data from file $f_i$;
            write evicted data to file $f_i$;
        end
    end
    $c - -$;
end
return;
```

of different websites are shaped into the same or similar pattern, which makes it difficult for an attacker to distinguish. To generate sufficient perturbations (i.e., dummy cache activities), our defense mimics the cache behaviour of a web browser loading websites by

(1) *Simulating dummy rendering process of a web browser with dummy I/O operations;*
(2) *Implementing with multiple processes*

4.1 Details of Cache Shaping

We simulate the dummy rendering process of a web browser by creating dummy I/O operations on a user's computer. These dummy I/O operations can create high cache activities, which will be sufficient to hide the real cache activities of a web browser. Specifically, we use `read()` and `write()` functions over dummy files stored on the disk as the vehicles to generate these dummy I/O operations. Our defense method can be described as below:

- Step 1: Cache shaping creates n dummy files on the local disk of a user's computer.
- Step 2: Cache shaping reads the dummy files from the local disk to prime the LLC. As a result, it evicts data previously cached in LLC. This step mimics the process when a web browser renders a website from local cached files.
- Step 3: Cache shaping writes the evicted data from the LLC in Step 2 to the dummy files on the local disk. This step mimics the caching process of a web browser.

Our defense initiates before a user visits a website. It runs in multiple processes, where each process runs independently. In each process, our defense repeats Step 2 and Step 3 based on a pre-defined counter c. Each process iterates all the memory pages in the entire LLC to repeat the dummy operations. Each memory page includes multiple cache sets and each cache set consists of multiple cache lines. The pseudo code of our defense in one process is described in Algo. 1. For the ease of description, we set the number of dummy files as the same as the number of memory pages in Algo. 1. The two parameters are not correlated and do not have to be the same in the implementation. We implement cache shaping in C/C++ with around 550 lines of code.

4.2 Defense Overhead Optimization

Our defense could generate relatively high CPU slowdowns due to repeated dummy I/O operations. We also explore two approaches that can reduce the potential high overheads in defense.

Random Switch. The main idea of random switch is to randomly turn on or off the dummy operations while cache shaping operates. Specifically, if we integrate cache shaping with random switch, our defense will still run in multiple processes and each process will still iterate the entire LLC independently. However, when each process iterates each memory page, it flips a coin and performs dummy I/O operations created by Step 2 and Step 3 in cache shaping with a probability of p. We denote this probability p as *perturbation-on probability*. This is a pre-defined parameter, which can be decided in advance. Randomly turning off the dummy operations at some memory pages can reduce the overall CPU slowdowns. The description of cache shaping in the previous subsection can be considered as a special case of cache shaping with random switch when perturbation-on probability is $p = 1$.

Attack Detection. As not every website in the real world is injected by the malicious JavaScript code running cache-based website fingerprinting, running our defense for every website that a user opens can cause unnecessary performance slowdowns. One proactive method is to detect the potential malicious website first when a user opens a website. If the detection indicates malicious, then the user can close the tab of the malicious website directly or enable cache shaping to continue to browser the website if needed. If the detection indicates benign, the user can keep cache shaping off to avoid performance slowdowns.

Detecting cache-based website fingerprinting is feasible as the malicious website interacts with the user's computer to derive cache activities, which leaves fingerprints on the user's computer. More specifically, the malicious website needs to measure cache activities of the LLC on the user's computer with a certain sampling rate. This measurement by the attacker periodically triggers fetching and evicting data on the LLC on the user's computer with a fixed interval. This creates a cache pattern that is distinguishable from the cache patterns of other benign websites.

Obviously, the user can record the cache activities by himself when he visits each website and formulates the detection as a *binary classification*. To record cache activities, the user can leverage the same method measuring the LLC used by the attacker to monitor his own LLC for detection purpose.

Details of Attack Detection. To build a classifier, our detection method will first collect some cache activities of websites (with and without cache-based website fingerprinting) in a web browser to train a binary classifier. When a user would like to visit a website and open it with a web browser, the detection method will record the cache activities of this website and pass the cache trace to the trained classifier to obtain a prediction. If the trained classifier indicates the website is malicious (i.e., positive) with a pre-defined confidence, then our proposed defense (cache shaping) will be on. Otherwise, cache shaping will remain off. The pre-defined confidence can be tuned to explore trade-offs in precision and recall in the detection. Running the detection does not need dummy I/O operations, which results in lower overheads than constantly running the defense, which complements the defense and optimize the overall performance slowdowns.

5 PERFORMANCE EVALUATION

5.1 Data Collection and Datasets

We collect large-scale datasets with multiple web browsers (Chrome, Firefox and Tor browser) and different OSs (Linux and Windows) for the evaluation of our study.

Specifically, we build a tool in Python to automatically collect cache activities of a web browser. Our tool integrates the JavaScript code published in [34] to measure cache activities with a certain sampling rate when a web browser loads a website. In addition, our tool utilizes `Selenium` library [4] to automatically launch a web browser and load the URL of each website one after another.

We build a blank webpage, which includes the malicious JavaScript code, and use it as the malicious website in our experiment. To facilitate the data collection in a large scale, we host the malicious website locally instead of hosting a malicious website remotely on a server. Note that this does not change the cache traces we collect, as either way, the malicious JavaScript code needs to be run locally on the user's machine to measure cache traces. During the data collection, as in the previous study [35], we use the two-tab scenario. One tab is the malicious website running the JavaScript code and the other tab is a monitored (or unmonitored) website. In our collection, we keep the response cache of a web browser on as most of the users do in the real world.

We examine three web browsers, including Chrome 85, Firefox 81, and Tor Browser 64, in the data collection. For each web browser, we use the same parameters from [35]. Specifically, if the web browser is Chrome or Firefox, we collect a cache trace of one website visit for 30 seconds. With a sampling rate of 500 Hz, it samples 15,000 measurements in a cache trace. For Tor Browser, as its timer resolution is only 100ms, instead of measuring the cache access latency, we count how many rounds the malicious code is able to iterate over the entire LLC within each time interval (e.g., 100ms). We use 10 Hz as the sampling rate and record cache activities for 50 seconds for each website visit in Tor browser. Thus, there are 500 measurements in each cache trace collected from Tor Browser.

We use a Linux machine (Ubuntu 20.04) with Intel Core i5-7500 processor, 16 GB RAM and 6 MB of Last Level Cache to collect cache traces. In addition, we also collect some cache traces on a Windows machine (Windows 10) with Intel Core i5-7500 processor, 16 GB RAM and 6 MB of Last Level Cache.

To perform the closed-world evaluation, we select 100 monitored websites and collect a number of 100 cache traces per website in one dataset. The 100 monitored websites are selected from Alexa[1] top 100. We first obtain four datasets for the closed-world evaluation when there is no defense. We denote them as

- Chrome100 (Linux), Chrome100 (Windows), Firefox100 (Linux), Tor100 (Linux).

Moreover, we choose another 5,000 unmonitored websites and collect 1 cache trace per website for each dataset. The 5,000 unmonitored websites are selected from Alexa top 100 to Alexa top 5100. We obtain four datasets when there is no defense.

- Chrome5000 (Linux), Chrome5000 (Windows), Firefox5000 (Linux), Tor5000 (Linux).

In addition to collecting datasets when there is no defense, we also collect corresponding datasets in different scenarios, including the defense is cache masking designed by [35], the defense is cache shaping (i.e., our defense), and the defense is cache shaping with different parameters, for different experiments in our evaluation. The overall size of the cache traces we collected is 16 GBs with more than 270,000 cache traces. The entire data collection lasted for five months, from August 2020 to January 2021.

[1]https://www.alexa.com/topsites

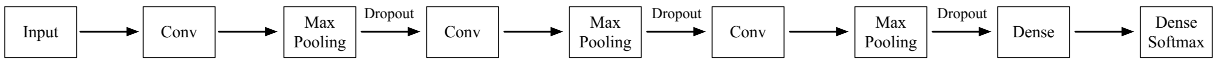


Figure 3: The architecture of CNN in this study.

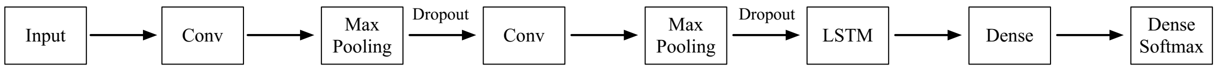


Figure 4: The architecture of LSTM in this study

Table 1: Tuned Hyperparameters for CNN and LSTM on Chrome100 (Linux)

Hyperparameters	Search Space	CNN	LSTM
Optimizer	{Adam, SGD, Adamax, Adadelta}	Adam	SGD
Learning Rate	{0.001, 0.003, ..., 0.01}	0.008	0.004
Decay	{0.00, 0.01, 0.02, ..., 0.90}	0.412	0.639
Batch Size	{16, 32, 64, 128, 256}	64	64
CNN Activation	{softsign, tanh, elu, selu, relu}	[tanh; relu; relu]	[tanh; elu;]
LSTM Activation	{softsign, tanh, elu, selu, relu}	–	[tanh; softsign;]
Dropout	{0.1, 0.2, ..., 0.7}	[0.5; 0.5; 0.2]	[0.1; 0.1]
Dense Layer Size	{100, 110, ...,200}	180	160
Convolution Number	{32, 64, 128, 256}	[32; 32; 32]	[16; 64]
Filter Size	{4, 6, ..., 26}	[14; 14;8]	[8; 14]
Pool Size	{1, 2, 3, 4,5,6,7}	[2; 4; 4]	[7; 7]
LSTM Units	{4,8,16,...,256}	–	128

5.2 Architectures of Neural Networks

We leverage Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks as two classifiers to evaluate the performance of attacks and defenses. The CNN consists of 3 convolutional layers, 3 pooling layers and a fully-connected layer (with softmax as the activation function). The LSTM includes 2 convolutional layers, 2 pooling layers and a LSTM layer. The architectures of CNN and LSTM are described in Fig. 3 and Fig. 4.

5.3 Evaluation Setting

Given a dataset, we randomly select 64% of data for training, 16% for validation and 20% for testing. We perform 5-fold cross validation. We implement the neural networks with Keras 2.2.4 as the front end and TensorFlow 1.15 as the backend. We train and test neural networks on a Linux machine (Ubuntu 18.04) with Intel Core i9-9900 processor, 64 GB RAM, and a Nvidia Titan RTX GPU.

We tune the hyperparameters of each neural network using NNI (Neural Network Intelligence) [3]. NNI is a open-source toolkit developed by Microsoft. During the hyperparameter tuning, we run at most 50 epochs or abort the tuning if the accuracy does not further improve after 10 consecutive epochs. After the tuning, we record the hyperparameters reported by NNI and train the classifiers locally with our machine. The tuned hyperparameters we derive for Chrome on Linux are presented in Table 1.

If the web browser or OS changes, we re-tune the hyperparameters. On the other hand, if the web browser and OS remain the same but it is defended data, we reuse the tuned hyperparameters. For instance, for defended datasets collected with Chrome and Linux, we use the tuned hyperparameters in Table. 1 for CNN and LSTM. In our evaluation, when we investigate performance over defended data, we always assume that an attacker can adapt to the defense.

Table 2: Closed-World Evaluation: Attack accuracy (mean ± standard deviation) on non-defended datasets

Non-defended dataset	CNN	LSTM
Chrome100 (Linux)	88.5% ± 1.1%	88.3% ± 1.0%
Firefox100 (Linux)	73.1% ± 0.9%	70.3% ± 1.1%
Tor100 (Linux)	36.1% ± 0.8%	35.1% ± 1.2%
Chrome100 (Windows)	83.5% ± 0.8%	87.8% ± 1.0%

In other words, we always retrain neural networks with defended data and then test with defended data.

5.4 Closed-World Evaluation

Experiment A.1: Validating the results of cache-based website fingerprinting. We validate the attack results of cache-based website fingerprinting over our datasets. As we can observe from Table 2, the attack can achieve very high accuracy when the web browser is Chrome or Firefox. For instance, given CNN as the classifier, cache-based website fingerprinting can achieve more than 88% accuracy. The attack accuracy over cache traces collected from Tor browser is only about 36% due to a much lower sampling rate in Tor Browser. The two classifiers, CNN and LSTM, obtain similarly results on the same dataset. The observations we obtain are consistent with the results reported in the previous study [35].

Experiment A.2: Impact of the number of measurements per trace. We also examine the impact of the number of measurements/samples in a cache trace on attack accuracy over non-defended datasets. This aspect, which was not examined in [35], can help us understand the attack better. In Fig. 5, we can observe that when we increase the number of measurements (i.e., the length of

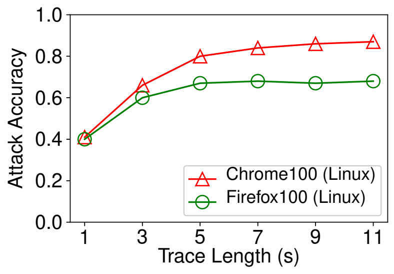


Figure 5: The impact of the number of measurements per trace on attack accuracy over non-defended datasets.

Table 3: Closed-World Evaluation: Attack accuracy (mean ± standard deviation) on defended datasets generated by cache masking.

Defended dataset	CNN	LSTM
Chrome100 (Linux)	72.1% ± 1.4%	77.1% ± 1.1%
Firefox100 (Linux)	55.0% ± 1.2%	51.6% ± 1.2%
Tor100 (Linux)	1.7% ± 0.2%	1.4% ± 0.1%
Chrome100 (Windows)	80.5% ± 1.3%	83.7% ± 1.1%

Table 4: Closed-World Evaluation: Attack accuracy (mean ± standard deviation) on defended datasets generated by cache shaping (with $m = 4$ processes and $n = 128$ dummy files)

Defended dataset	CNN	LSTM
Chrome100 (Linux)	13.2% ± 1.4%	12.6% ± 1.2%
Firefox100 (Linux)	25.5% ± 1.3%	18.1% ± 1.0%
Chrome100 (Windows)	14.4% ± 1.2%	17.6% ± 1.5%

a trace) in a cache trace, the attack accuracy increases, particularly within the first 5 seconds.

Experiment A.3: Reproducing the results of cache masking. We leverage the implementation of cache masking in [35] and collect defended data generated by it. We still use CNN and LSTM as the classifiers and retrain the neural networks over defended data. As we can observe from Table 3, although the attack accuracy decreases compared to the results in Table 2, an attacker can still achieve high accuracy over defended data collected from Chrome and Firefox. This suggests that cache masking is not effective when an attacker adapts the defense. On the other hand, the attack accuracy over defended data from Tor Browser is low and close to 1% of random guess. The observations we have are consistent with the ones reported in [35]. Our results in the Experiment A.1 and A.3 indicate that we successfully reproduced the results in [35].

Experiment A.4: Comparison between Our Defense and Cache Masking. We evaluate the efficacy of our defense against cache-based website fingerprinting and also compare it with cache masking. As cache masking is able to reduce the attack accuracy to the level of random guess for Tor Browser as shown in the previous experiment, we focus on the comparisons over defended data from Chrome and Firefox.

We first implement our defense with C/C++ and collect cache traces generated by our defense with the same monitored websites using Chrome and Firefox respectively. In this experiment, we choose the number of processes as $m = 4$ and the number of dummy files as $n = 128$ in our defense. We defer the discussions on the impact of the number of processes and the number of dummy files in later experiments. Each dummy file is initialized with a cache-line size (64 bytes). We set the maximum size of each dummy file as 1 MB to hold evicted data. When the size of a dummy file reaches 1 MB, we reset it to the cache-line size to hold additional evicted data. We retrain the two neural networks over defended data.

Table 5: CPU slowdown (geometric mean) between cache masking and cache shaping.

	Cache Masking [35]	Cache Shaping ($m = 4$, $n = 128$)
Integer	20.8%	57.4%
Floating	24.3%	52.1%

Defence Efficacy. As we can see from Table 4, our defense can significantly reduce the attack accuracy even if an attacker adapts the defense and retrains the classifiers. Specifically, if we use CNN as the classifier, our defense can reduce the attack accuracy to 13% on Chrome100 (Linux) dataset and 25% on Firefox100 (Linux) dataset. Compared to 72% and 55% derived by cache masking in the last experiment, our defense obviously renders much better protection against cache-based website fingerprinting attacks.

Performance Slowdown. In addition to the attack accuracy, we also compare the CPU slowdowns between our defense and cache masking. Specifically, we leverage SPEC CPU 2017 benchmarks [5] to measure the performance slowdown to a user's computer, where the slowdown is caused by our defense or cache masking. SPEC CPU 2017 is an industry-standardized tool for measuring CPU performance with 43 benchmarks. Among the 43 benchmarks, we measure the performance slowdowns primarily with 10 integer rate benchmarks and 13 floating point rate benchmarks. The CPU slowdowns are evaluated in geometric mean.

As illustrated in Table 5, cache masking introduces 20.8% slowdown on integer benchmarks and 24.3% slowdown on floating point benchmarks. On the other hand, our method introduces 57.4% slowdown on integer benchmarks and 52.1% slowdown on floating point benchmarks when the number of processes is $m = 4$ and the number of dummy files is $n = 128$. A more detailed comparison on each specific benchmark is presented in Fig. 6. Our defense introduces a higher slowdown than cache masking, which is expected. This is because our defense needs to introduce more dummy operations to improve the efficacy of the defense. This is a necessary trade-off.

Experiment A.5: The Impact of The Number of Processes in Our Defense (Closed-World). We investigate the impact of the number of processes in our defense. Specifically, we select the number of processes as $m = \{1, 2, 4, 8\}$ respectively, and collect the defended data produced by our defense based on each value of m. For each value of m, we retrain the two neural networks and record the corresponding accuracy. The number of dummy files remains $n = 128$ in this experiment.

As we can see from Table 6, if we increase the number of processes in our defense, it can further mitigate the attack accuracy. For example, given CNN as the classifier, the attack accuracy can be reduced to 10.9% over Chrome100 (Linux) if we increase the

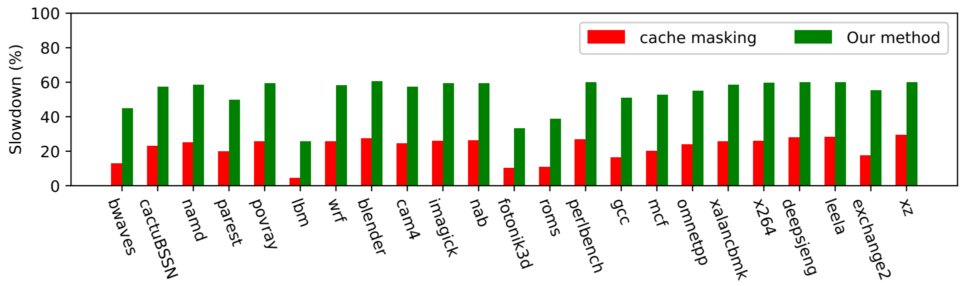


Figure 6: CPU slowdown comparison between cache masking and our method ($m = 4$ and $n = 128$) according to SPEC CPU 2017 benchmarks (the first 13 on the left are floating point rate benchmarks and the rest are integer rate benchmarks).

Table 6: Closed-World Evaluation: The impact of the number of processes on attack accuracy (mean) on defended datasets generated by cache shaping ($n = 128$).

Defended dataset	Classifier	No. of Processes m			
		1	2	4	8
Chrome100 (Linux)	CNN	43.1%	38.7%	13.2%	10.9%
	LSTM	40.4%	36.3%	12.6%	6.0%
Firefox100 (Linux)	CNN	31.0%	28.2%	25.5%	11.2%
	LSTM	31.5%	21.1%	18.1%	7.0%

Table 7: CPU slowdown (geometric mean) in cache shaping with SPEC CPU 2017 benchmarks. ($m = \{1, 2, 4, 8\}$, $n = 128$)

	No. of Processes m			
	1	2	4	8
Integer	51.4%	70.0%	57.4%	71.8%
Floating	42.6%	63.6%	52.1%	66.9%

number of processes to 8. On the other hand, when the number of processes increases, in general, the performance slowdown overall increases as shown in Table 7. For example, when $m = 8$, the performance slowdown will increases to 71.8% and 66.9%. Note that when $m = 2$, the performance slowdown is higher than the one with $m = 4$. We were not able to figure out what caused this outlier in our experiment.

Experiment A.6: The Impact of The Number of Dummy Files in Our Defense (Closed-World). We investigate the impact of dummy files in our defense. Specifically, we choose the number of dummy files as $n = \{2, 8, 32, 128\}$ and set the number of processes as $m = 4$, and we collect the defended data produced by our defense. For each value of n, we retrain the two neural networks and record the corresponding accuracy. As shown in Table 8, increasing the number of dummy files can improve the efficacy of our defense. For instance, given CNN as the classifier and $n = 2$, the attack accuracy is 23.5% over Chrome100 (Linux) defended dataset. When n increases to 128, the attack accuracy is further reduces to 13.2%.

In Table 9, we can also observe that the CPU slowdown, overall, remains at the same level when we change the number of dummy files. This is expected as the dummy files are stored on disk, which do not directly affect the performance of a CPU. On the other hand, the disk storage that our defense needs linearly increases with the number of dummy files, which is a necessary trade-off.

Table 8: Closed-World Evaluation: The impact of the number of dummy files on attack accuracy (mean) on defended datasets generated by cache shaping ($m = 4$).

Defended dataset	Classifier	No. of dummy files n			
		2	8	32	128
Chrome100 (Linux)	CNN	23.5%	20.3%	19.3%	13.2%
	LSTM	27.1%	11.0%	12.0%	12.6%

Table 9: CPU slowdown (geometric mean) in cache shaping with SPEC CPU 2017 benchmarks. ($n = \{2, 8, 32, 128\}$, $m = 4$)

	No. of dummy files n			
	2	8	32	128
Integer	59.2%	63.1%	57.1%	57.4%
Floating	53.4%	57.5%	53.1%	52.1%
Storage	2 MBs	8 MBs	32 MBs	128 MBs

Table 10: Attack accuracy (mean) and CPU slowdown (geometric mean) of cache shaping with random switch ($m = 4$, $n = 128$). Attack accuracy is evaluated on Chrome100 (Linux) defended dataset with CNN as the classifier.

	Perturbation-on Prob. p			
	0.25	0.5	0.75	1.0
Accuracy	39.1%	28.0%	17.1%	13.2%
Integer	49.1%	53.6%	57.1%	57.4%
Floating	43.0%	46.3%	50.4%	52.1%

Experiment A.7: Cache Shaping with Random Switch. We examine the performance of our defense by integrating it with random switch. As described in Sec. 4, the main idea of random switch is to randomly turn our defense on at each memory page while our defense iterates all the memory pages. Specifically, random switch turns our defense on at each memory page with obfuscation-on probability p. We choose $p = \{0.25, 0.5, 0.75, 1\}$ in this experiment. We choose $m = 4$ and $n = 128$ for all the values of p. Note that, when $p = 1$, it is our defense without random switch.

As we can see from Table 10, when the perturbation-on probability increases, our defense produces higher perturbations overall, and therefore the attack accuracy decreases. On the other hand, a lower perturbation-on probability produces lower perturbations, which leads to lower CPU slowdowns.

Table 11: Open-World Evaluation: precision and recall over non-defended datasets.

Monitored (Non-defended)	Unmonitored (Non-defended)	Classifier	Tuned for Precision		Tuned for Recall	
			Precision	Recall	Precision	Recall
Chrome100 (Linux)	Chrome5000 (Linux)	CNN	99.0%	88.2%	98.5%	98.5%
		LSTM	99.9%	83.1%	99.9%	83.1%
Firefox100 (Linux)	Firefox5000 (Linux)	CNN	98.0%	62.6%	94.7%	85.8%
		LSTM	99.8%	61.8%	99.7%	82.4%
Tor100 (Linux)	Tor5000 (Linux)	CNN	99.0%	27.0%	94.3%	60.4%
		LSTM	99.5%	41.2%	99.0%	60.2%

Table 12: Open-World Evaluation: precision and recall over defended datasets generated by cache shaping with $m = 4$ and $n = 128$.

Monitored (Defended)	Unmonitored (Defended)	Classifier	Tuned for Precision		Tuned for Recall	
			Precision	Recall	Precision	Recall
Chrome100 (Linux)	Chrome5000 (Linux)	CNN	85.3%	1.5%	85.3%	1.5%
		LSTM	99.8%	4.6%	97.1%	69.7%
Firefox100 (Linux)	Firefox5000 (Linux)	CNN	98.9%	2.2%	99.7%	54.8%
		LSTM	99.5%	21.7	68.1%	68.0%
Tor100 (Linux)	Tor5000 (Linux)	CNN	0%	0%	0%	0%
		LSTM	0%	0%	0%	0%

5.5 Open-World Evaluation

Experiment B.1: Attack Results over Non-Defended Data. We first examine the performance of the attack over non-defended data. We report the detailed attack results of precision and recall over non-defended data in Table 11. In general, the two classifiers achieve high precision and recall over Chrome and Firefox in the open-world evaluation. For example, CNN obtains 99.0% precision with 88.2% recall over Chrome datasets if we tune the threshold for the best precision and 98.5% precision with 98.5% recall when we tune the threshold for the best recall. Compared to Chrome and Firefox, the open-world evaluation is less effective over Tor datasets. This is expected as the sampling rate and the number of measurements in a cache trace in Tor Browser are much lower than the ones in the other two web browsers. The precision-recall curve of attack results over non-defended data is shown in Fig. 7.

Experiment B.2: Attack Results on Defended Data Generated by Our Method. We examine the efficacy of cache shaping in the open-world setting. Specifically, we collect the defended data generated by cache shaping corresponding to the datasets we examined in the previous experiment. When we collect the defended data, we set the number of processes as $m = 4$ and the number of dummy files as $n = 128$ in cache shaping.

The precision-recall curve of attack results over defended data generated by our method is shown in Fig. 8. As we can observe, our method cache shaping is very effective against the attack in the open-world setting, where our method can significantly reduce either precision or recall, or both. More detailed results are presented in Table 12.

Experiment B.3: Comparison between Our defense and Cache Masking. We compare the defense efficacy of cache shaping with cache masking in the open-world setting. Specifically, we leverage the defended datasets produced by our defense from the last experiment and collect the corresponding defended datasets generated by cache masking. We use CNN as the classifier and

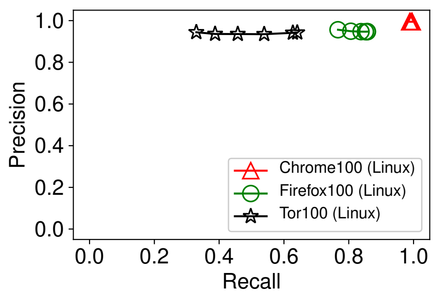


Figure 7: Precision-recall curves over non-defended datasets (classifier: CNN).

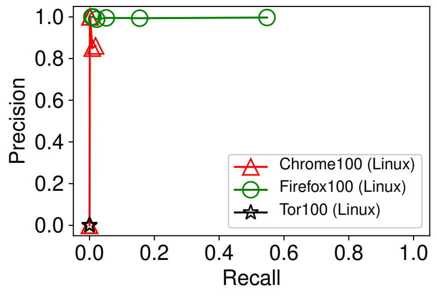


Figure 8: Precision-recall curves over defended datasets generated by cache shaping with $m = 4$ and $n = 128$ (classifier: CNN).

examine Chrome100 (Linux) and Chrome5000 (Linux) defended datasets.

As shown in Fig. 9, the precision-recall curve of the cache-based website fingerprinting attack over defended data generated by cache masking is very close to the precision-recall curve of the attack over non-defended data. This is suggests that cache masking is also not effective in the open world evaluation when an attacker adapts

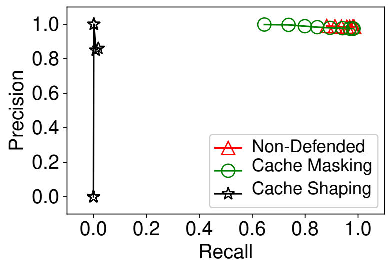


Figure 9: Comparison of cache shaping and cache masking in precision-recall curves (classifier: CNN; Chrome100 (Linux) and Chrome5000 (Linux) defended datasets).

and retrains a classifier with defended cache traces. On the other hand, our defense cache shaping is much effective compared to cache masking according to the precision-recall curve.

5.6 Attack Detection

As mentioned in Sec. 4, a user could detect whether a website is malicious by measuring the cache activities himself. This is because, compared to benign websites, a malicious website leaves unique cache patterns due to its attack behaviors. *The detection can be formulated as a binary classification.* To demonstrate the efficacy of our detection, we consider the following two scenarios for an attacker:

Scenario 1 (Basic Content). A website contains the malicious JavaScript code and some basic content (e.g., a small image or/and a short text). This scenario carries less cache activities from the website itself, which leads to less noise in the attack detection. The attacker can opt to turn on or off the malicious JavaScript code. If the code is on, the website is considered as malicious. If the code is off, the website is considered as benign.

We add a small image (about 500 KB) or/and short text to our blank webpage including the malicious JavaScript code, and use it as the website in this scenario. We collect 1,000 cache traces when the malicious JavaScript is on and 1,000 cache traces when the malicious JavaScript code is off. We use a different image or a different text on the webpage when we collect each cache trace to mimic a (slightly) different website. We use Chrome as the web browser, Linux as the OS, and each cache trace lasts for 30 seconds with a sampling rate of 500Hz. We denote this dataset as *Basic1000*.

Scenario 2 (Comprehensive Content). We assume that a real-world website is compromised by the attacker, where the malicious JavaScript code is injected to the website. This scenario carries more cache activities from the website itself, which makes the attack detection more challenging. The attacker can opt to turn on or off the malicious JavaScript code. If the code is on, the website is considered as malicious. Otherwise, it is considered as benign.

However, since we do not have control of any of the these real-world websites to inject the malicious JavaScript code in our lab setting, we simulate this scenario with the following way. We open two tabs in a web browser synchronously, where one tab opens a website from Alexa top websites and the other tab opens our blank webpage with the malicious JavaScript code. With this setup, we simulate the cache activities when a real-world website running

Table 13: Precision and recall of attack detection

Dataset	Tuned for Precision		Tuned for Recall	
	Precision	Recall	Precision	Recall
Basic1000	98.5%	99.9%	98.5%	99.9%
Alexa1000	98.0%	99.9%	98.0%	99.9%

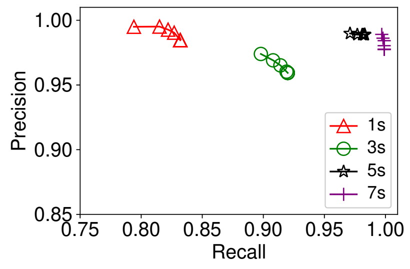


Figure 10: The impact of measurement time on attack detection over Alexa1000 dataset (classifier: CNN)

the malicious JavaScript code. We collect one cache trace when the malicious JavaScript is on and one cache trace when the malicious JavaScript code is off for each website from Alexa top 1000 websites. We use Chrome as the web browser, Linux as the OS, and each cache trace lasts for 30 seconds with a sampling rate of 500Hz. We denote this dataset as *Alexa1000*.

Experiment C.1: Performance of the Attack Detection. We evaluate the performance of our attack detection with Basic1000 and Alexa1000 respectively. We use our CNN as the classifier in the detection. Given a dataset, we select 64% of data for training, 16% for validation and 20% for testing. As the detection is a binary classification, we use precision and recall as the metrics. As we can see from Table 13, in each dataset, our detection can detect malicious websites with high precision and high recall based on cache activities.

Experiment C.2: Impact of the Measurement Time on Attack Detection. Although we obtain high precision and recall in the last experiment, we assume that the detection can monitor cache activities for 30 seconds. In the real world, a user may want to detect a malicious website as soon as possible to minimize the negative impact on user experience. Therefore, we investigate when our detection has a much shorter measurement time (i.e., has a much smaller number of features in each trace), what impacts it may have on the performance of detection.

Specifically, we examine Alexa1000 dataset when each trace only has measurements for 1, 3, 5, and 7 seconds respectively, and retrain the classifier. The comparison of the corresponding precision-recall curves is summarized in Fig. 10. The figure indicates that the detection needs to run for about 5 seconds to achieve extremely high precision and high recall. However, even with 1 second, the detection can still achieve 98.5% precision and 83.2% recall.

Experiment C.3: Performance of the Attack Detection over Unseen Websites. We further examine the performance of our detection over unseen websites to test how robust our detection is against cache-based website fingerprinting attacks. Specifically, we assume that the training data and test data for detection are not from the same dataset.

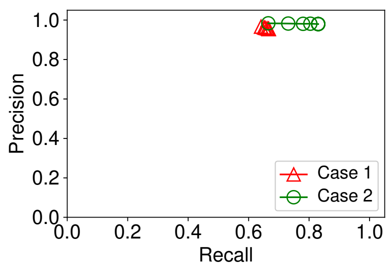


Figure 11: Precision and recall of attack detection over unseen websites (Case 1: training with Basic100 and test with Alexa100; Case 2: training with Alexa100 and test with Basic100).

More specifically, we investigate two cases, where Case 1 takes training data from Basic1000 but selects test data from Alexa1000 and Case 2 takes training data from Alexa1000 but selects test data from Basic1000. As we can see from Fig. 11, in this more challenging setting, the detection remains effective to some degree (i.e., ≥95% precision and ≥63% recall), but is not as effective as the results presented in the previous experiment. For instance, when the detection has a lower recall, malicious websites may not be detected and still reveal user privacy.

The above detection performance drop is expected, as it is challenging for a machine learning model to achieve extremely good performance when there is a significant discrepancy between training data and test data. Note that this is not a special limitation to our detection but is a general challenge to machine-learning-based methods. Increasing the size of the training data can mitigate this problem. However, it will take longer time to collect data and train our attack detection model, especially considering there are millions of websites in the real world. This suggests that we should use the detection to complement our proposed defense rather than completely relying on the detection alone.

6 DISCUSSIONS AND FUTURE WORK

Mitigating CPU slowdowns. As the original cache pattern of each website is distinguishable due to the unique content of each website in the real world, perturbing all the cache activities to the same or similar pattern as in our defense inevitably introduces high CPU slowdowns. This is a natural trade-off in privacy-preserving techniques, where higher perturbations generally preserve more privacy against attackers but cause high overheads.

Seeking new ways to further mitigate CPU slowdowns will be important. For instance, similar as some of the existing defenses (e.g., Walkie-Talkie [41]) against traffic-based website fingerprinting, generating cache activities such that a pair of two websites are indistinguishable might be able to reduce defense overheads rather than making the cache patterns of all the websites the same/similar in our current defense.

However, that requires knowing cache activities of websites in advance in order to pair two websites with similar cache pattern together. This is not scalable or easy to maintain as cache activities can change frequently due to content change on websites in the real world. On the other hand, our current design does not require the prior knowledge of cache activities of websites, which is easier to deploy. Identifying more important features (i.e., measurements) in traces and primarily perturb these features only might also be helpful to reduce the overhead in defense [13, 18]. We will leave it in future work.

Attacking across different settings. Similar as the majority of studies in website fingerprinting, we assume that the attacker has the same setting (i.e., same web browser, same OS, same LLC-size) as a user when we evaluate the attack accuracy and defense efficacy in this study. However, a real-world attacker may not have the exact same setting as a user. In other words, there will be discrepancy between the training data collected by the attacker and the test data captured from a user. The discrepancy affects the attack accuracy derived from a classifier. Techniques such as *transfer learning* could be utilized to address the problem in future work. For instance, recent studies [37, 39] have shown the advantages of transfer learning in addressing the cross-setting problem in the context of traffic-based website fingerprinting.

7 RELATED WORK

Traffic-based website fingerprinting. Many studies [12, 14, 15, 19, 24, 27, 28, 31, 36–38, 40] leverage encrypted network traffic to infer which website a user visits. With deep neural networks as classifiers, recent research [24, 31, 36] are able to achieve extreme high accuracy. Wang et al. [39] achieved high attack accuracy over only few samples by leveraging transfer learning based on triplet networks. Dani and Wang [9] utilized semantic correlation of the content of webpages to improve attack accuracy over multiple traces. To defend against traffic-based website fingerprinting attacks, many defenses [10, 12, 13, 16, 36, 41] have also been proposed to perturb the pattern of encrypted traffic. Li et al. [18] proposed to leverage feature importance to locate the most important features (or packets) in order to reduce bandwidth overheads introduced by defenses. Several recent studies [23, 30, 33] also explore how to generate adversarial examples of encrypted traffic traces to fool website fingerprinting attacks that rely on deep neural networks.

Cache-based website fingerprinting. Oren et al. [25] show that Prime+Probe can be launched remotely by using malicious JavaScript code through a web browser. They show that it is feasible to carry out cache-based website fingerprinting with high sampling rates. Shusterman et al. [34, 35] proposed to measure the cache activities of the entire LLC to perform website fingerprinting with relatively low sampling rates. Cronin et al. [8] optimized the attacking technique from [34, 35] to fingerprint websites visited via ARM devices. Their experimental results show that the cache occupancy channel can also be utilize to compromise user privacy on those newly released devices with ARM processors.

To mitigate the threat of Prime+Probe through JavaScript, major web browsers reduced the resolution of the time function in JavaScript which can be used by an attacker to distinguish a cache hit and a cache miss [2, 32]. For example, the precision of Performance.now() is decreased to 0.1 ms in Chrome and 1 ms in Firefox.

Chen et al. [7] proposed to defend against cache-based attacks by using features provided by hardware transactional memory. As these defenses focus on protecting specific cache lines rather than cache activities of the entire LLC, they are not suitable for defending against cache-website fingerprinting attacks using cache occupancy as indicated by [34, 35].

8 CONCLUSIONS

We propose a new defense which can perturb cache activities of a web browser, and therefore, defend against cache-based website fingerprinting. Compared to previous methods, our defense remains effective even if an attacker is aware of the defense and retrains classifiers. Extensive experiments over large-scale datasets collected from major web browsers and operating systems are carried out in order to validate the efficacy of our proposed defense. Moreover, we also explore methods that can mitigate the CPU slowdowns caused by our defense.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. The authors were partially supported by National Science Foundation (CNS-1947913).

REFERENCES

[1] [n.d.]. Cache Shaping Defense. https://github.com/UCdasec/CacheShaping
[2] [n.d.]. Mozilla Foundation. Security advisory 2018-01. https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/
[3] [n.d.]. NNI: An open source AutoML toolkit for neural architecture search and hyper-parameter tuning. https://github.com/Microsoft/nni
[4] [n.d.]. Selenium automates browsers. https://www.selenium.dev/
[5] [n.d.]. SPEC CPU 2017 Benchmarks. https://www.spec.org/cpu2017/
[6] Jo. Booth. 2015. Not So Incognito: Exploiting Resource-Based Side Channels in JavaScript Engines. *Bachelor's thesis, Harvard College* (2015).
[7] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. 2018. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proc. of ACM ASIACCS'15*.
[8] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An Exploration of ARM System-Level Cache and GPU Side Channels. In *Proc. of ACSAC,21*.
[9] Jimmy Dani and Boyang Wang. 2021. HiddenText: Cross-Trace Website Fingerprinting over Encrypted Traffic. In *Proc. of IEEE Conference on Information Reuse and Integration for Data Science (IEEE IRI'21)*.
[10] Wladimir De la Cadena, Asya Mitseva, Jens Hiller, Jan Pennekamp, Sebastian Reuter, Julian Filter, Thomas Engel, Klaus Wehrle, and Andriy Panchenko. 2020. TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting. In *Proc. of ACM CCS'20*.
[11] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Trans. Archit. Code Optim.* 8 (2012). Issue 4.
[12] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2012. Peek-a-Boo, I still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proc. of IEEE S&P'12*.
[13] Jiajun Gong and Tao Wang. 2020. Zero-delay Lightweight Defenses against Website Fingerprinting. In *Proc. of USENIX Security'20*.
[14] J. Hayes and G. Danezis. 2016. K-Fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *Proc. of USENIX Security'16*.
[15] Dominik Hermann, Rolf Wendolsky, and Hannes Federrath. 2009. Website Fingertinging: Attacking Popular Privacy Enhancing Tehnologies with the Multinomial Naive-Bayes Classifier. In *Proc. of ACM Workshop on Cloud Computing Security*.
[16] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. 2016. Toward an Efficient Website Fingerprinting Defense. In *Proc. of ESORICS'16*.
[17] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi. 2020. NetCAT: Practical Cache Attacks from the Network. In *Proc. of IEEE S&P'20*.
[18] Haipeng Li, Ben Niu, and Boyang Wang. 2020. SmartSwitch: Efficient Traffic Obfuscation Against Stream Fingerprinting. In *Proc. of SecureComm'20*.
[19] Marc Liberatore and Brian Neil Levine. 2006. Inferring the Source of Encrypted HTTP Connections. In *Proc. of ACM CCS'06*.
[20] J. Liedtke, H. Hartig, and M. Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proc. of 3rd IEEE Real-Time Technology and Applications Symposium*.
[21] F. Liu and R. B. Lee. 2014. Random Fill Cache Architecture. In *Proc. of 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*.
[22] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapotre, and G. Gogniat. 2018. Run-time Detection of Prime + Probe Side-Channel Attack on AES Encryption Algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*.
[23] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2020. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. In *Proc. of USENIX Security'21*.
[24] Se Eun Oh, S. Sunkam, and N. Hopper. 2019. p-FP: Extraction, Classification, and Predication of Website Fingerprints. In *Proc. of Privacy Enhancing Technologies (PETS'19)*.
[25] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *in Proc. of ACM CCS'15*.
[26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *topics in Cryptology – CT-RSA 2006*.
[27] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Penekamp, K. Wehrle, and T. Engel. 2016. Website Fingerprinting at Internet Scale. In *Proc. of NDSS'16*.
[28] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. 2011. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proc. of Workshop on Privacy in the Electronic Society*.
[29] M. K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proc. of 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*.
[30] Mohammad Saidur Rahman, Mohsen Imani, Nate Mathews, and Matthew Wright. 2021. Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks with Adversarial Traces. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1594–1609.
[31] V. Rimmer, D. Preuveneers, M. Juarez, T. V. Goethem, and W. Joosen. 2018. Automated Website Fingerprinting through Deep Learning. In *Proc. of NDSS'18*.
[32] M. Schwarz, Moritz Lipp, and D. Gruss. 2018. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *Proc. of NDSS'18*.
[33] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y. Zhao. 2021. A Real-time Defense against Website Fingerprinting Attacks. In *Proc. of the 14th ACM Workshop on Artificial Intelligence and Security (AISec'21)*.
[34] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. 2020. Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality. *IEEE Transactions on Dependable and Secure Computing* 18 (2020), 2042–2060.
[35] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *Proc. of USENIX Security'19*.
[36] P. Sirinam, M. Imani, M. Juarez, and M. Wright. 2018. Deep Fingerprinting: Understanding Website Fingerprinting Defenses with Deep Learning. In *Proc. of ACM CCS'18*.
[37] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright. 2019. Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-shot Learning. In *Proc. of ACM CCS'19*.
[38] Jean-Pierre Smith, Prateek Mittal, and Adrian Perrig. 2021. Website Fingerprinting in the Age of QUIC. In *Proc. of PETS'21*.
[39] Chenggang Wang, Jimmy Dani, Xiang Li, Xiaodong Jia, and Boyang Wang. 2021. Adaptive Fingerprinting: Website Fingerprinting over Few Encrypted Traffic. In *Proc. of ACM CODASPY'21*.
[40] Tao Wang, Xiang Cui, Rishab Nithyannand, Rob Johnson, and Ian Goldberg. 2014. Effective Attacks on Proable Denfenses for Website Fingerprinting. In *Proc. of USENIX Security'14*.
[41] T. Wang and I. Goldberg. 2017. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *Proc. of USENIX Security'17*.
[42] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *Proc. of the 34th annual international symposium on Computer architecture (ISCA'07)*.
[43] Yuval Yarom. [n.d.]. Mastik: A micro-architectural side-channel toolkit. ([n. d.]). https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf