NetShaper: A Differentially Private Network Side-Channel Mitigation System

Amir Sabzi, Rut Vora, Swati Goswami, Margo Seltzer, Mathias Lécuyer, Aastha Mehta
The University of British Columbia

Abstract

The widespread adoption of encryption in network protocols has significantly improved the overall security of many Internet applications. However, these protocols cannot prevent *network side-channel leaks*—leaks of sensitive information through the sizes and timing of network packets. We present NetShaper, a system that mitigates such leaks based on the principle of traffic shaping. NetShaper's traffic shaping provides differential privacy guarantees while adapting to the prevailing workload and congestion condition, and allows configuring a tradeoff between privacy guarantees, bandwidth and latency overheads. Furthermore, NetShaper provides a modular and portable tunnel endpoint design that can support diverse applications. We present a middlebox-based implementation of NetShaper and demonstrate its applicability in a video streaming and a web service application.

1 Introduction

With the proliferation of TLS and VPN, traffic encryption has become the de facto standard for securing data in transit in Internet applications. Traffic can be encrypted at various layers, such as HTTPS, QUIC, and IPSec. While these protocols prevent *direct* data breaches on the Internet, they cannot prevent leaks through *indirect* observations of the encrypted traffic.

Indeed, encryption cannot conceal the shape of an application's traffic, i.e., the sizes, timing, and number of packets sent and received by an application. In many applications, these parameters strongly correlate with sensitive information. For instance, traffic shape can reveal video streams [55], website visits [14, 66], the content of VoIP conversations [69], and even users' medical and financial secrets [19].

In such *network side-channel leaks*, an adversary (e.g., a malicious or a compromised ISP) observes the shape of an application's traffic as it passes through a link under its control and infers the application's sensitive data from this shape.

Obfuscation techniques, which add ad hoc noise [44] or adversarial noise [6, 30, 34, 49, 51, 56] in an application's network traces, do not provide comprehensive protection against network side-channel attacks [73]. In fact, recent advances in machine learning (ML) have greatly improved the ability to filter out noise due to congestion or path variations and infer secrets from noisy data [14, 31, 55, 58]. For instance, our own novel classifier based on Temporal Convolution Networks (TCN) [9] can infer video streams even from short bursts of noisy measurements over the Internet (see §2 for details). Alternatively, a sensitive application could try to improve side-channel resilience by splitting traffic over multiple network paths [22, 65] or by using dedicated physical links that are not controlled by the adversary. However, such solutions are inadequate against a powerful adversary that can monitor a large fraction of the Internet [13] and may incur prohibitive network administration costs for small users on the Internet.

In contrast, a principled and practical approach to mitigating network side-channel leaks is *traffic shaping*. It involves modifying the victim's packet sizes and timing to make the resultant shape independent of secrets, so that an adversary cannot infer the secrets despite observing the (shaped) traffic.

Constant shaping involves sending fixed-sized packets at a constant rate, which is secure but incurs non-trivial bandwidth and/or latency overhead for applications with variable or bursty workloads [54]. Variable shaping strategies attempt to adapt traffic shapes to reduce the overhead at the cost of some privacy. However, the state-of-the-art (SOTA) variable shaping strategies rely on ad hoc heuristics that yield weak privacy guarantees [50, 66, 67] or unbounded privacy leaks [16, 17, 29, 36, 43]. Some techniques provide strong guarantees but require extensive a priori profiling of an applications' traffic to compute shapes [45, 73].

In addition to a shaping strategy, network side-channel mitigation also requires a robust implementation of packet padding and transmit scheduling. Many solutions attempt to protect traffic by controlling shaping from only one end of a communication (i.e., either a client or a server) and provide only best-effort protection [20, 44, 59]. Other solutions rely on trusting third-party mediators (e.g., Tor bridges), which implement shaping between the clients and mediators and between the servers and mediators [48, 70]. In Pacer [45], both applica-

tion endpoints integrate a shaping system to comprehensively mitigate network side channels. However, Pacer encumbers application end hosts with non-trivial changes in the network stack to implement shaping, thus deterring adoption.

In this work, we address two main questions. First, is there a variable traffic shaping strategy that provides quantifiable and tunable privacy guarantees at runtime without requiring extensive pre-profiling of application traffic? Second, can traffic shaping be provided as a generic, portable, and efficient solution that can be integrated in different network settings and can support diverse applications?

We present NetShaper, a network side-channel mitigation system that answers both questions in the affirmative. First, NetShaper relies on a differential privacy (DP) based traffic shaping strategy, which provides quantifiable and tunable privacy guarantees. NetShaper specifies the privacy parameters for a configurable window of transmission. Moreover, it can configure the parameters independently for each direction of traffic on a communication link. The DP guarantees can be composed based on these parameters to achieve bounded privacy leaks for arbitrary bidirectional traffic. Overall, applications can tune the shaping based only on the privacy guarantees they desire and the overheads they can afford, without the need for profiling their traffic. While strong privacy guarantees require DP parameters that incur large overheads, in practice, NetShaper can defeat SOTA attacks with even small amounts of DP noise, thus incurring low overheads.

Secondly, we present a traffic shaping tunnel with a modular endpoint design that can conceptually be integrated with any network stack and within any node. The tunnel implements padding and transmit scheduling of packets while adhering to the DP guarantees *by design*. By placing the tunnel endpoint in a middlebox at the edge of the private network, NetShaper can simultaneously protect the traffic of multiple applications. Moreover, the middlebox can amortize the shaping overheads among multiple flows without compromising the privacy for individual flows.

Together, the DP shaping strategy and NetShaper's tunnel provide effective network side-channel mitigation for diverse applications, such as video streaming and web services, with modest overheads. To the best of our knowledge, NetShaper is the first system to provide dynamic traffic shaping with quantifiable and tunable privacy guarantees based on DP.

Contributions. (i) We design a new attack classifier based on a Temporal Convolution Network (TCN) [9] and demonstrate its ability to infer videos streams from traffic shapes under noisy network traffic measurements in the Internet (§2). (ii) We model network side-channel mitigations as a differential privacy problem and provide a traffic shaping strategy that offers (ε,δ)-differential privacy guarantees (§3). (iii) We design a QUIC-based traffic shaping tunnel and present a middlebox-based implementation of the tunnel, which supports traffic shaping while adhering to DP guarantees (§4). (iv) We demonstrate NetShaper's efficacy in defeating a SOTA classifier [55] and our new TCN classifier. We empirically evaluate the tradeoffs between NetShaper's privacy guarantees and performance overheads while mitigating network side-channel leaks in two classes of applications that have already been used in prior work, namely video streaming and web service (§5). (vi) We present a formal proof of NetShaper's differential privacy guarantees (§C).

2 Background, Motivation, and Overview

2.1 Network Side-Channel Attacks

We start by explaining the workings of a network side-channel attack with an example application. Consider MedFlix, a fictitious medical video service that offers videos on symptoms, treatment procedures, and post-operative care. The goal of an adversary is to infer the videos streamed by users visiting the service. ISPs can aggregate such information to build per-user profiles and subsequently monetize them. Additionally, competitors might exploit network side channels to acquire corporate intelligence without detection.

The adversary performs the attack in two stages. First, the adversary requests each video from the medical service as a client and collects traces of the bidirectional network traffic generated while streaming each video. The adversary may collect multiple traces for each video stream to account for network variations in the traffic. The adversary then builds a classifier over the captured traces to identify the video streams. Prior work has used several features for classification, such as packet sizes, inter-packet timing, total bytes transferred in a burst of packets, the burst duration and inter-burst interval, and the direction of packets or bursts [55].

We reproduce the Beauty and the Burst (BB) classifier [55], a state-of-the-art CNN classifier for classifying video streams from network traces. Furthermore, we present a new TCN classifier [9], which is an improvement over the BB classifier. We describe the classifiers in §A. Here, we evaluate the efficacy of the two classifiers for a network side-channel attack.

We set up a video service and a video client as two Amazon AWS VMs placed in Oregon and Montreal, respectively. The video server hosts a dataset of 100 YouTube videos at 720p resolution with MPEG-DASH encoding [68]. The client streams the first 5 min of each video over HTTPS and collects the resulting network packet traces using tcpdump. We stream each video 100 times, thus collecting a total of 10,000 traces.

The classifiers' goal is to predict the video from a network trace. For each classifier, we transform each packet trace into a sequence of burst sizes transmitted within 1s windows and normalize the sequence by dividing each burst size by the total size of all bursts. We evaluate the performance of both classifiers with three datasets: a small dataset consisting of 20 videos with their 100 traces each (i.e., total 2000 traces), a medium dataset with 40 videos (4000 traces), and a large dataset comprising all 100 videos (10000 traces). We train the

classifiers for 1000 epochs with an 80-20 train-test split. BB's classification accuracy, recall, and precision with the small dataset are 0.85, 0.85, and 0.78, respectively, which drop to 0.61, 0.63, and 0.49, respectively, with the medium dataset, and further drop to 0.01 each for the large dataset. TCN's accuracy, recall, and precision are above 0.99 for all datasets. TCN performs better than BB because it is a complex model with residual layers and, hence, is robust to noise in the traces.

Similarly, advanced ML classifiers are capable of identifying web traffic [14, 58]. In general, classifiers will continue to evolve, increasing the adversary's capabilities to make inferences from noisy measurements. Hence, we need principled mitigations that address current SOTA attacks and achieve quantifiable leakage, which can be configured based on privacy requirements and overhead tolerance.

2.2 Key Ideas

A secure and practical network side-channel mitigation system must satisfy the following design goals: **G1.** Mitigate leaks through all aspects of the shape of transmitted traffic, **G2.** Provide quantifiable and tunable privacy guarantees for the communication parties, **G3.** Minimize overheads incurred while guaranteeing privacy, **G4.** Support a broad class of applications, and **G5.** Require minimal changes to applications.

NetShaper's shaping prevents leaks of the traffic content through sizes and timing of packets transmitted along each direction between application nodes (G1). In addition, NetShaper relies on the following three key ideas.

Differentially private shaping. Unlike constant shaping, variable shaping can adapt traffic shape, potentially based on runtime workload patterns, and thus significantly reduce shaping overheads. Unfortunately, existing variable shaping techniques either have unbounded privacy leaks, offer only weak privacy guarantees, or require extensive profiling of an application's network traces. NetShaper's novel differential privacy (DP) based shaping strategy provides quantifiable and tunable bounds on privacy leaks, without relying on profiling of application traffic (G2). NetShaper shapes an application's traffic in periodic, fixed-length *shaping intervals* and provides DP in the length of the application byte stream (burst) accumulated within each interval. The DP guarantees compose over a sequence of multiple intervals and, thus, for streams of arbitrary length (albeit with degraded guarantees).

Shaping in a middlebox. NetShaper uses a tunnel abstraction to implement traffic shaping. The tunnel shapes application traffic such that an adversary observing the tunnel traffic cannot infer application secrets. In principle, a tunnel endpoint could be integrated with the application host (e.g., in a VM isolated from the end-host application) or in a separate node through which the application's traffic passes. NetShaper relies on the second approach and implements the tunnel endpoint as a middlebox, which could be integrated with an existing network element, such as a router, a VPN gateway, or a firewall. The middlebox implementation enables securing multiple applications without requiring modifications on individual end hosts (G4). Furthermore, it allows pooling multiple flows with the same privacy requirements in the same tunnel, which helps to amortize the per-flow overhead (G3).

Minimal modifications to end applications. By default, NetShaper shapes all traffic through a tunnel with a fixed differential privacy guarantee. However, an application can explicitly specify different DP parameters to adapt the privacy guarantee enforced for its traffic, as well as bandwidth and latency constraints and any prioritization preferences on a per-flow basis. This requires only a small modification in the application; it must transmit a shaping configuration message to the middlebox. Thus, NetShaper offers a balance between being fully application-agnostic and optimizing for privacy or overhead with minimal support from applications (G2, G5).

2.3 Threat Model

NetShaper's goal is to hide the content of an application's network traffic. Hiding the type of traffic [57], the communication protocol [70], or the application identity [21, 27] are non-goals, although NetShaper can adapt its shaping strategy to address these goals. The applications are non-malicious and do not leak their own secrets.

We assume that the application endpoints are inside separate trusted private networks (e.g., each node is behind a VPN gateway node) and the adversary cannot infiltrate the private network, or the clients and servers within it. (Thus, we exclude covert attacks [72] and colocation-based attacks [45, 55]). The adversary controls network links in the public Internet (e.g., ISPs) and can record, measure, and tamper with the victim application's traffic as it traverses the links under the adversary's control. The adversary can precisely record the traffic shape – the sizes, timing, and direction of packets – between the gateway nodes. In particular, it may have access to observations of arbitrary known streams to train its attack. It may also have knowledge about NetShaper, including its shaping strategy and privacy configurations.

We do not consider threats due to observing the IP addresses of packets [32], although NetShaper can hide IP addresses of applications behind a shared traffic shaping tunnel.

NetShaper does not address leaks of one application's sensitive data through the traffic shape of colocated benign applications. Such leaks can arise, for instance, due to microarchitectural interference among applications colocated on a host or among their flows if they pass through shared links. End hosts could implement orthogonal mitigations against colocated applications [15, 39, 45, 63] and combine NetShaper's shaping with TDMA scheduling on network links [12, 64].

We present a middlebox-based NetShaper implementation that can be integrated with an organization's trusted gateway router. NetShaper's trusted computing base (TCB) includes all components in the organization's private network and the

middleboxes. Bugs, vulnerabilities or side channels in the middleboxes that threaten traffic confidentiality could be mitigated using orthogonal techniques, such as software fault isolation [61], resource partitioning [42], and constant-time implementation techniques [7].

Under these assumptions, NetShaper prevents leaks of application secrets through the sizes and timing of packets transmitted in either direction between the application endpoints.

2.4 A Primer on Differential Privacy

Finally, we provide a brief primer on DP, the steps involved in building a DP mechanism, and the key properties of DP that are relevant in the context of traffic shaping (§3).

Developed originally for databases, DP is a technique to provide aggregate results without revealing information about individual database records. Formally, a randomized algorithm $\mathcal{M}$ is (ε,δ)-DP if, for all $\mathcal{R} \subseteq \text{Range}(\mathcal{M})$ and for all *neighboring* databases d, d' that differ in only one element:

$$P[\mathcal{M}(d) \in \mathcal{R}] \leq e^{\varepsilon} P[\mathcal{M}(d') \in \mathcal{R}] + \delta \qquad (1)$$

The parameter ε represents the *privacy loss* of algorithm $\mathcal{M}$, i.e., given a result of $\mathcal{M}$, the information gain for any adversary on learning whether the input database is d or d' is at most e^{ε} [37]. The δ is the probability with which $\mathcal{M}$ fails to bound the privacy loss to e^{ε}.

Building such a randomized DP algorithm $\mathcal{M}$ involves three main steps: (i) defining neighboring database states, (ii) defining a database query and determining the sensitivity of the query to changes in neighboring databases, and (iii) adding noise to the query. Neighboring databases d and d', as mentioned above, are characterized by the *distance* between the databases, which quantifies the granularity at which the DP guarantee applies. Traditionally, this distance is defined as the number of records that differ between d and d'. However, DP also extends to other neighboring definitions and distance metrics used in specific settings [18, 41].

Given a database query q, the sensitivity of the query Δq is the max difference in the result achieved when the query is executed on the neighboring databases d and d'. Intuitively, a larger Δq implies higher probability of an adversary inferring from a result the database on which the query was executed, thus incurring higher privacy loss. To mitigate this privacy loss, a DP mechanism therefore adds noise to the query result to hide the true result and the underlying database. Popular noise mechanisms are Laplace [26, §3.3] and Gaussian [23].

DP provides three properties. As we will show in §3.2, these are also of relevance to NetShaper's DP traffic shaping. First, DP is resilient to post-processing: given the result r of any (ε,δ)-DP mechanism $\mathcal{M}$, any function $f(r)$ of the result is also (ε,δ)-DP. As a result, any computation or decision made on a DP result is still DP with the same guarantees. Second, DP is closed under adaptive sequential *composition*: the combined result of two DP mechanisms $\mathcal{M}_1$ and $\mathcal{M}_2$ is also DP, though with higher losses (ε and δ). We use the Rényi-DP definition [47] to achieve simple but strong composition results and subsequently convert the results back to the standard DP definition. Third, DP is robust to auxiliary information: the guarantee from Equation 1 holds regardless of any side information known to an attacker. Therefore, the attacker's knowledge of the shaping mechanism does not affect its privacy guarantees. That is, an attacker knowing or controlling part of the database cannot extract more knowledge from a DP result than without this side information.

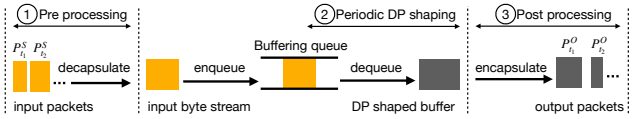


Figure 1: Overview of DP shaping

3 Differentially Private Traffic Shaping

The goal of differentially private shaping is to dynamically adjust packet sizes and timing based on the available data stream, while ensuring that the DP guarantees hold for any information that an adversary (§2.3) can observe.

We first formally model an application's input stream as a packet sequence $S = \{P_1^S, P_2^S, P_3^S, \dots\}$, where $P_i^S = (l_i^S, t_i^S)$ indicates that the i^{th} input packet in S has length l_i^S bytes and is transmitted at timestamp t_i^S. We call the total duration of a finite stream $\tau^S \triangleq \max t_i^S - t_1^S$. Without shaping, an adversary can precisely observe S and infer the content, which is correlated with the stream (§2.1).

Figure 1 provides a high-level overview of the differentially-private shaping mechanism. Shaping happens in periodic intervals of fixed length T, called the *DP shaping intervals*. (1) As the packets in an application's stream arrive, the payload bytes extracted from the packets are placed into a *buffering queue*, Q. (2) In each periodic interval, the DP shaping algorithm performs a *DP query*: it measures the length of Q with DP, to determine the number of bytes to transmit in the next interval. NetShaper then prepares a DP shaped buffer using the payload bytes from Q, and additional dummy bytes if required. This shaped buffer is enqueued to be sent over the network at the end of the DP shaping interval, right before the next interval starts. (3) Finally, data in the shaped buffer may be split into one or more packets, as part of a post-processing step, and transmitted to the network.

The size of each shaped buffer generated in an interval has $(\varepsilon_T, \delta_T)$-DP guarantees. The per-interval guarantees compose over a sequence of multiple intervals, thus providing DP guarantees for traffic streams of arbitrary lengths. The guarantees degrade as the stream length increases (Prop 2)

We now discuss the steps of building a DP mechanism for traffic streams in §3.1 and the privacy guarantees in §3.2.

3.1 DP for Traffic Streams

We now discuss the three steps for building a DP mechanism on traffic streams. Specifically, we present our neighboring definition for streams, define the DP query on streams that NetShaper runs and bounds its sensitivity, and show the DP mechanism that we use to make the query DP.

Step 1: Neighboring Definition. Building a DP mechanism requires a suitable definition of neighboring streams, which requires a notion of distance between streams. The neighboring definition has two implications. First, it determines the sensitivity for a "query" subject to the DP mechanism and, in turn, the amount of noise necessary for ensuring a given DP guarantee. Second, it defines the granularity of privacy guarantee: intuitively, neighboring streams are indistinguishable based on only the results of the DP mechanism applied to them. Hence, we aim for a neighboring definition for which "streams have many neighbors", and the sensitivity of the DP query we want to run is as low as possible.

Defining neighbors over streams has two challenges. First, the streams can be arbitrarily long and the distance between streams would typically increase with stream length. Secondly, if we consider streams with packet timestamps at the finest granularity, the distance between the streams may be as large as the sum of sizes of all packets in both streams. Both these factors imply that a stream would either have few neighbors (enabling weak privacy), or the neighboring definition would need to specify a large distance threshold, requiring a lot of noise for strong DP guarantees.

To keep the neighboring distance threshold small, we take two steps. First, we define the notion of a *neighboring window*, which is a time interval of configurable length W over input streams. For notational convenience, we set W as an integral multiple of the DP shaping interval T and write $W = kT$, although this is not a strict requirement.

Secondly, to measure the distance between streams over a neighboring window, we consider the total bytes in each stream (burst lengths) transmitted within coarse-grained time intervals and use the difference in the burst lengths in the intervals within the window. Intuitively, we need an interval granularity that is coarse enough to generate similar burst length sequences in more streams, but is also small enough to bound the accumulation of differences over time (to bound sensitivity, our next step). As will become clear with Prop 1, the DP shaping interval T is the coarsest granularity that we can use. Hence, considering a neighboring window starting at a timestamp t_w, i.e., $[t_w,\ t_w + W)$, we define the following representation of stream S over the window $[t_w,\ t_w + W)$ at granularity T: $S_{t_w,W} = \{L^S_{t_w}, L^S_{t_w+T}, L^S_{t_w+2T}, \ldots, L^S_{t_w+(k-1)T}\}$, where $L^S_t \triangleq \sum_{(l^S_i, t^S_i)\in S} \mathbb{1}\{t^S_i \in [t, t+T)\} l^S_i$ is the total application bytes accumulated in Q in the interval $[t, t+T)$.

We now present the following neighboring definition:

Definition 1. *Two streams S and S' are neighbors if, for any neighboring window $[t_w,\ t_w + W)$, the L1-norm distance between their representations $S_{t_w,W}$ and $S'_{t_w,W}$ is less than Δ_W. Formally, S and S' are neighbors if:*

$$\max_{t_w} \left\| S_{t_w,W} - S'_{t_w,W} \right\|_1 \leq \Delta_W. \tag{2}$$

We utilize the L1-norm (the sum of absolute values) to quantify the distance between two traffic streams, as it captures differences in both traffic size and temporal alignment, at the granularity of T. We will show (in Prop 1) that despite our restriction of defining neighboring streams at a granularity of T, and of computing distances over windows of length W, we can quantify NetShaper's DP guarantees for streams at any granularity and of any length.

The neighboring window length W and neighboring distance Δ_W are both configuration parameters, which are set before the start of an application's transmission. In practice, W would be in the order of milliseconds to seconds. Subsequently, one would determine Δ_W based on the typical difference of traffic between application streams over windows of length W. The practical upper bound for Δ_W is the NIC's line rate times window length W, but smaller values based on domain knowledge are often possible.

Step 2: DP query and sensitivity. In NetShaper, a DP query measures the buffering queue length L with DP at intervals T. This noisy measurement determines the number of bytes that must be transmitted in the interval. To make the measurement of L differentially private, we need to bound the sensitivity Δ_T of the queue length variable L. This sensitivity Δ_T is the maximum difference in L that can be caused by changing one application stream to neighboring one. Formally, consider two alternative neighboring streams S and S' passing through the queue. Suppose that when transmitting S (similarly S'), the queue length at time k is denoted by L_k (respectively L'_k). Then, assuming w.l.o.g. that $k \geq k'$:

$$\Delta_T = \max_{k=0}^{\tau} \max_{S,S'} |L_k - L'_k| \tag{3}$$

Bounding Δ_T is still challenging though, as our neighboring definition only bounds the difference in traffic between two streams over any window of length upto W. Because of this, when $\tau >> W$, differences between what S and S' would enqueue in Q can accumulate over time, and the difference betwen L_t and L'_t can grow unbounded over time.

To bound Δ_T, NetShaper relies on the key assumption that the tunnel can always transmit all incoming data from application streams within any W-sized time window. That is,

Assumption 1. *All bytes enqueued prior to or at time t are transmitted by time $t + W$.*

To enforce this assumption, NetShaper implements a time to live in the buffering queue, flushing all bytes older than W from Q (see §4 for more details). Intuitively, this assumption caps the accumulation of traffic differences in Q to the maximum difference over W, i.e., Δ_W. Since the size of Q

cannot differ by more than Δ_W when changing a stream by a neighboring one, the difference between DP query results of Q under two neighboring streams—which is the sensitivity of the DP query, Δ_T—is upper-bounded by Δ_W. Formally:

Proposition 1. *NetShaper enforces* $\Delta_T \leq \Delta_W$.

Proof sketch. Consider any two streams S and S', as in Equation 3, and any measurements time k. The proof proceeds in two steps. First, under Assumption 1, streams can accumulate queued traffic for at most W, so two different streams can create a difference $|L_k - L'_k|$ of at most Δ_W. Second, dequeuing can only make two different queues closer: Consider query time k, with queue lengths $L_k > L'_k$ (the opposite case is symmetric). For a DP noise draw z, we have $\tilde{L}_k > \tilde{L}'_k$. Since shaping sends at least as much data under $\tilde{L}_k$ as under $\tilde{L}'_k$, but no more than $\tilde{L}_k - \tilde{L}'_k$, after dequeuing we have $|L'_{k+1} - L_{k+1}| \leq |L'_k - L_k|$. In summary, the queue difference under two different streams Δ_T can grow to at most Δ_W due to data queuing, and dequeuing only decreases that difference, and hence $\Delta_T \leq \Delta_W$. The complete proof is in §C. □

Step 3: Adding Noise. With sensitivity bounded at Δ_W, we can now query L with DP using an additive noise mechanism, which entails sampling noise z from a DP distribution and computing the DP buffer queue length as $\tilde{L}_k \triangleq L_k + z$. Specifically, NetShaper uses the Gaussian mechanism, in which the noise z is sampled from a centered normal distribution $z \sim \mathcal{N}(\mu, \sigma^2)$, where the variance is parameterized by ε_T, δ_T, and Δ_W: $\sigma^2 = (2\Delta_W^2)/(\varepsilon_T^2)\ln(1.25/\delta_T)$. Parameters ε_T, δ_T determine the amount of noise added to each DP query result.

3.2 Privacy Analysis

The previous section defines $(\varepsilon_T, \delta_T)$-DP guarantees for the traffic transmitted in an individual shaping interval. We now discuss (i) the guarantees for longer application streams, (ii) the guarantees on a packet-level sequence derived from a shaped buffer sequence, and (iii) the privacy implications for streams that fall outside of the neighboring definition.

Guarantees for streams. Recall from the previous section that shaping happens at intervals T; in each interval we perform a DP query on the buffering queue length L and create a shaped buffer of length $\tilde{L}$, which is subsequently queued for transmitting over the network at the end of the interval. Enqueueing data into the shaped buffer at the end of each shaping interval creates a sequence of states for the shaped buffer $\{(\tilde{L}_1, T), (\tilde{L}_2, 2T), (\tilde{L}_3, 3T), \dots\}$. Although this sequence is not technically observable by an adversary, this is where we prove our DP guarantees using DP composition over queries performed during the stream transmission.

Proposition 2. *For any stream S of duration $\tau^S \leq \tau$, NetShaper enforces (ε, δ)-DP for the sequence $\{(\tilde{L}_1, T), (\tilde{L}_2, 2T), (\tilde{L}_3, 3T), \dots\}$, with $\varepsilon, \delta \triangleq$ DP_compose$(\varepsilon_T, \delta_T, \lceil \frac{\tau}{T} \rceil)$.*

Proof. Consider two neighboring streams S and S'. By design, the times at which shaped buffers are queued are independent of application data, so $\{(\tilde{L}'_1, T'), (\tilde{L}'_2, 2T'), (\tilde{L}'_3, 3T'), \dots\} = \{(\tilde{L}_1, T), (\tilde{L}_2, 2T), (\tilde{L}_3, 3T), \dots\}$, and we can restrict our considerations to the sequences $\{\tilde{L}_1, \tilde{L}_2, \tilde{L}_3, \dots\}$ and $\{\tilde{L}'_1, \tilde{L}'_2, \tilde{L}'_3, \dots\}$. By Prop. 1, the sensitivity of each measurement is at most Δ_W. By the Gaussian DP mechanism, the measured queue size $\tilde{L}$ in each interval is $(\varepsilon_T, \delta_T)$-DP. Using DP composition over the $\lceil \frac{\tau}{T} \rceil$ DP queries made during any duration τ yields the (ε, δ)-DP guarantee. □

We use Rényi-DP composition on the Gaussian mechanism for DP_compose(). NetShaper provides a DP guarantee for streams of any length t, with the guarantee degrading gracefully as DP composition (of order $\sqrt{\tau}$ as the length grows).

Guarantees for packet sequences. Data from the shaped buffer is sent over the network, and transmitted as a packet sequence denoted by $O = \{P_1^O, P_2^O, P_3^O \dots\}$. These packets are a post-processing of the DP-shaped buffer. As long as the packets are generated independently of any secret data, they preserve the DP guarantees of shaping due to the post-processing property of DP. This yields the following result, directly implied by Prop 2 and DP post-processing:

Corollary 1. *For any stream S of duration $\tau^S \leq \tau$, NetShaper enforces (ε, δ)-DP for its output packet sequence O, with $\varepsilon, \delta \triangleq$ DP_compose$(\varepsilon_T, \delta_T, \lceil \frac{\tau}{T} \rceil)$.*

Privacy for non-neighboring streams. Finally, if the distance between two streams is larger than Δ_W, e.g., say $k\Delta_W$ for some factor k, NetShaper still provides a (degraded) DP guarantee through group privacy [26, Theorem 2.2] applied to the Gaussian mechanism. Namely, a DP query in each shaping interval T for the non-neighboring stream provides $(k\varepsilon_T, \delta_T)$-DP, and the guarantees can be extended for the stream duration τ by applying DP composition to this new value.

Interpretation of the guarantees. NetShaper's shaping algorithm provides a $(\varepsilon_T, \delta_T)$-DP guarantee on the volume of application traffic enqueued (in the buffering queue) in a fixed-length interval (and, by post-processing, the volume of traffic observable on the network). Under perfect timing for the DP shaping interval T, the DP guarantee ensures that two neighboring streams, i.e., their contents, are indistinguishable with the probability as defined in Equation 1. Smaller ε_T and δ_T implies higher noise in shaping, which increases the uncertainty about the original stream in the shaped traffic.

Secondly, to enforce the DP guarantees, the DP noise of the Gaussian mechanism does not depend on the total number of flows. Intuitively, the DP guarantee is shared among all the streams multiplexed through the buffering queue simultaneously for a fixed amount of noise, and the overhead (i.e., noise added) gets amortized among the streams.

Privacy vs Performance. NetShaper's shaping mechanism introduces several parameters which impact privacy and performance, specifically latency and bandwidth overheads.

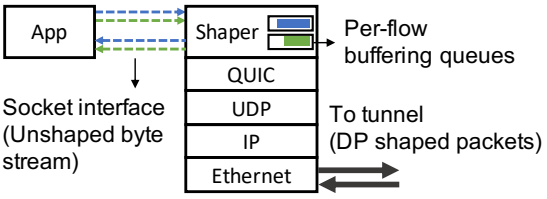


Figure 2: Overview of tunnel design (one endpoint)

These parameters include ε_T, δ_T, Δ_W, W, and T. Larger Δ_W requires lower ε_T for stronger privacy, which implies noisier measurements. Noisy measurements imply more overhead—when the noise is positive, dummy bytes need to be sent, incurring higher bandwidth overhead; when the noise is negative, fewer bytes are sent and the unsent bytes accumulated in the buffering queue incur a latency overhead. This is the privacy-overhead trade-off we expect from DP.

Additionally, the parameters T and W have a more subtle impact on the privacy-overhead trade-off. Traffic is shaped in intervals of T; thus, T impacts the latency and burstiness of the traffic. A smaller T provides lower transmission latency and smaller bursts per interval. However, it also requires more DP queries and, thus, incurs higher privacy loss when transmitting the complete stream of a given length τ. In practice, one would set T to the maximum value that can minimize privacy loss while providing tolerable latency.

A large W for a fixed Δ_W implies that neighboring streams can differ by at most Δ_W over a longer window size W, which weakens the neighboring definition and, hence, the privacy guarantees. While smaller W is desirable, the lower bound is T. Recall that, to bound Δ_T (Assumption 1), NetShaper must drop any bytes left in the buffering queue for longer than W. Since data leaves the buffering queue in each shaping interval after a DP query, setting $W = T$ would lead to immediate dropping of the bytes from the buffering queue that aren't transmitted in response to the DP query. This would happen in each interval where the DP query samples a negative noise value. These data drops would degrade NetShaper's performance in terms of both throughput and latency. Hence, we need $W >> T$ to ensure that data has time to leave the buffering queue before it is too old. Typically, one would set W based on application domain knowledge, such as the maximum size of a web request or the fact that videos often consist of a sequence of segments requested at 5s intervals.

We analyze the impact of different choices for these parameters on the privacy guarantees and overheads in §5.

4 Traffic Shaping Tunnel

A tunnel must address three requirements. First, it must satisfy DP guarantees. For this, the tunnel must complete DP queries and prepare shaped packets within each interval, and it must be able to transmit all payload bytes generated from an application within a finite window length (as defined in the DP strategy). Secondly, the payload and dummy bytes in the shaped packets must be indistinguishable to an adversary. For this, the payload and dummy bytes must be transmitted through a shared transport layer so that they are identically acknowledged by the receiver and subject to congestion control and loss recovery mechanisms. Finally, the tunnel must provide similar levels of reliability, congestion control, and loss recovery as expected by the application.

Figure 2 shows the design of one endpoint of NetShaper's traffic shaping tunnel. A similar endpoint is deployed on the other end of the tunnel. The shape of the traffic in the tunnel can be configured independently in each direction. The privacy loss in bidirectional streams is the DP composition of the privacy loss in each direction.

A tunnel endpoint consists of a shaping layer (Shaper) on top of QUIC, which in turn runs on top of a standard UDP stack. The tunnel endpoints establish a bidirectional QUIC connection and generate DP-sized transmit buffers in fixed intervals, which carry payload bytes from one or more application flows. In the absence of application payload, a tunnel endpoint transmits dummy bytes, which are discarded at the other endpoint. QUIC encrypts all outbound packets.

NetShaper adopts a transport-layer proxy architecture: each application terminates a connection with its local tunnel endpoint. The application byte stream is sent to the remote application over three piecewise connections: (i) between the application and its local tunnel endpoint, (ii) between the tunnel endpoints, and (iii) between the remote tunnel endpoint and the remote application. This ensures only one active congestion control and reliable delivery mechanism in the tunnel and that all bytes are subject to identical mechanisms[1].

The application and the tunnel endpoint shown in Figure 2 could either be colocated on the same host or located on separate hosts. In each case, the traffic between the application and the tunnel endpoint is assumed to be unobservable to an adversary. Our design (§4.1) does not distinguish between the two configurations. Our implementation (§4.2) assumes that the tunnel endpoint is located on a separate middlebox. We discuss security in §4.3 and alternate deployments in §4.4.

4.1 Tunnel Design and Operations

Tunnel setup and teardown. Before applications can communicate with each other, a NetShaper tunnel must be set up between their local tunnel endpoints. The initiator application sends a configuration message to its local tunnel endpoint with the source and destination IP addresses and ports, a reliability flag, and a privacy descriptor. The reliability flag indicates if the tunnel should provide reliable delivery semantics or not.

[1]We discard tunneling TCP through TCP as it causes TCP meltdown [4, 33], or TCP through UDP as it is unsafe. (TCP between the application hosts would retransmit lost payload bytes only, not any dummy bytes injected between the tunnel endpoints, making the dummy bytes observable.)

The privacy descriptor indicates the DP parameters to be used for shaping the tunnel traffic.

Upon receiving a configuration message, the Shaper establishes a QUIC connection with the remote tunnel endpoint and configures the reliability semantics and privacy parameters for each direction. It also initializes three types of bidirectional streams in the tunnel: control, dummy, and data streams. One *control stream* is used to transmit messages related to the establishment and termination of a connection between the application endpoints. A *dummy stream* transmits padding in QUIC packets in the form of STREAM frames[2]. The tunnel pre-configures a finite number of data streams, which carry payload bytes from one or more application flows.

When the tunnel is inactive for a period of time, one of the tunnel endpoints initiates a termination sequence and closes all open QUIC streams and the tunnel connection.

Connection establishment and termination. Once a tunnel is ready, applications can establish and terminate connections with each other, which is mediated by the tunnel. When the initiator application runs a connection establishment handshake with its local tunnel endpoint, the Shaper maps the application flow to a per-flow buffering queue and one of the inactive QUIC data streams in the tunnel, and notifies the remote tunnel endpoint. The remote tunnel endpoint establishes a connection with the receiver application and maps the receiver application's flow with the data stream. The connection termination handshake is handled similarly by the tunnel endpoints. The messages for connection establishment and termination are transmitted over the control stream in the tunnel and shaped according to the tunnel's parameters.

Outbound traffic shaping. The Shaper accumulates the outbound bytes of an application flow in a buffering queue before it transmits them in packets whose sizes and timing follow a distribution that guarantees DP. Within a tunnel, the Shaper transmits bytes from all active flows into a differentially-private packet sequence. At periodic intervals, called DP shaping intervals, it performs a DP query on the per-flow queues to determine the number of bytes $\tilde{L}$ to be transmitted according to the tunnel's DP parameters. It prepares a *shaped buffer* consisting of R payload bytes and D dummy bytes, where R is the minimum of $\tilde{L}$ and the application bytes available in the buffering queues, and $D = \tilde{L} - R$, which may lie between 0 and $\tilde{L}$. The Shaper then passes the buffer with the position and length of the padding to QUIC.

QUIC transforms the shaped buffer into one or more STREAM frames based on the congestion window, the flow window of the receiver endpoint, and the MTU (maximum transmission unit). It places the padding bytes into a dummy STREAM frame. QUIC packages the frames into packets, whose length is at most MTU minus the length of the headers and whose payload is encrypted. QUIC forwards the packets to the UDP layer, which subsequently transmits the prepared packets as quickly as it can, given the line rate of the NIC.

[2]We do not use QUIC's PADDING frames as they do not elicit acknowledgements and hence are distinguishable from STREAM frames [35].

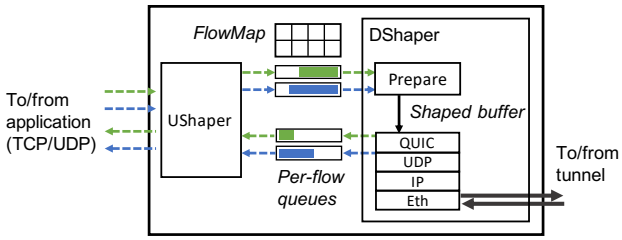


Figure 3: NetShaper middlebox design

To enforce Assumption 1, the Shaper tracks the expiry time of each byte enqueued in Q based on the arrival time and the neighboring window length W configuration. The Shaper drops the untransmitted bytes in the queue upon their expiry.

Inbound traffic processing. A tunnel endpoint receives shaped packets from the tunnel and applies inverse processing on each packet. QUIC receives the packet and sends an ACK to the sender. Subsequently, it decrypts the packet, discards the dummy frame, and forwards the payload bytes from the remaining STREAM frames to the application.

4.2 Middlebox Implementation

We present a middlebox-based NetShaper implementation. For ease of implementation, our prototype requires applications to explicitly connect to the middleboxes. In principle, NetShaper can transparently proxy application connections.

In our implementation (see Figure 3), a middlebox consists of two userspace processes. The `UShaper` mediates *unshaped* traffic between the applications and the middleboxes. The `DShaper` handles *DP shaped* traffic within the tunnel.

UShaper. The `UShaper` implements a transport server (or client) for interfacing with each local client (or server, respectively) application. For managing multiple flows, it shares a `FlowMap` table with the `DShaper`, which consists of an entry for each end-to-end flow. Each entry maps the piecewise connections with a pair of transmit and receive queues to carry the local application's byte stream, and shaping configurations (e.g., privacy descriptor) provided by an application at the time of flow registration.

The `UShaper` receives the outbound traffic from a sender application and enqueues the byte stream into a per-flow transmit queue shared with the `DShaper`. It also dequeues bytes from a per-flow receive queue, repackages them into transport packets and sends them to the receiver application.

DShaper. The `DShaper` consists of a `Prepare` thread and a QUIC worker thread. The `Prepare` thread instantiates a QUIC client/server to establish a tunnel with the remote middlebox and implements the DP shaping logic. On the transmit side, `Prepare` prepares shaped buffers based on DP queries on the transmit queues and then submits shaped buffers to the QUIC worker for transmission. On the receive size, the QUIC worker transmits ACK frames to the sender and then

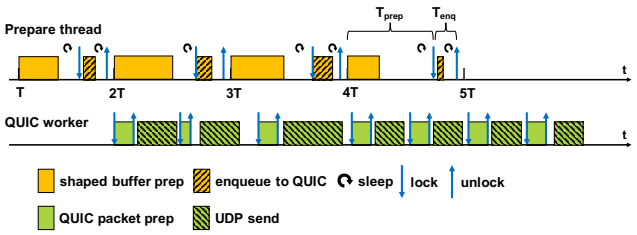


Figure 4: DShaper schedule

decrypts the QUIC packets, extracts the STREAM frames, and copies bytes (including dummy bytes) from each frame into the appropriate per-flow receive queue.

Ensuring secret-independent shaping. To enforce DP guarantees, DShaper should *transmit* exactly $\tilde{L}$ bytes in each DP shaping interval T. This would require ensuring: **P1.** Prepare computes $\tilde{L}$, allocates and prepares a buffer of length $\tilde{L}$, and passes the buffer to the QUIC worker within T, **P2.** the QUIC worker prepares encrypted packets from the buffer and sends them to UDP, such that the total payload size of the QUIC packets prepared in T is $\tilde{L}$, and **P3.** the UDP stack transmits packets totaling to $\tilde{L}$ payload bytes to the NIC in T. Enforcing all these properties would require a constant-time implementation for each step, which is non-trivial, or a strict time-triggered schedule for each step, which would significantly reduce link utilization and increase packet latencies.

Thanks to DP post processing, however, it suffices to ensure the property P1, and **P4.** that the QUIC worker transforms the shaped buffer into network packets independently of the application data. No other constraint on the sizes and timing of network packets is required to preserve DP.

Satisfying P1 involves one challenge. Although the application is physically isolated from the middlebox, its flow control behavior could be secret-dependent and could affect the middlebox's execution. For instance, the presence or absence of payload traffic from an application can affect the time DShaper requires to prepare the shaped buffers.

Thus, DShaper satisfies P1 as follows (see Figure 4 for reference). First, Prepare guarantees that $\tilde{L}$ is computed with a DP query in each interval. Secondly, Prepare guarantees that a shaped buffer of length $\tilde{L}$ is *prepared* within a fixed time T_{prep} within each interval. Thirdly, Prepare locks the shaped buffer for a fixed time, T_{enq}, during which it enqueues the buffer for a QUIC worker. This ensures that the buffer is completely enqueued before QUIC starts transmitting it and that QUIC receives the buffer only at fixed delays.

We empirically profile the time taken by Prepare for preparing and enqueueing shaped buffers for various DP lengths. We set T_{prep} and T_{enq} to maximum values determined from profiling, and T to the sum of these maximum values, i.e., T_{max}. If Prepare takes time less than T_{prep} (or T_{enq}, respectively) to prepare (or enqueue) a shaped buffer, it sleeps until the end of the interval before moving to the next phase.

To satisfy P4, Prepare and QUIC worker threads run on separate cores sharing only the shaped buffers. UShaper runs on yet a different core and shares the FlowMap and the per-flow transmit queues containing unshaped traffic only with Prepare. It shares the per-flow receive queues with the QUIC worker, but they contain only shaped frames from the QUIC worker. Finally, we assume that QUIC encrypts and decrypts shaped buffers in constant-time. With this, the execution of the QUIC worker becomes independent from Prepare and secret-independent overall. Consequently, the QUIC worker and the UDP stack can packetize the shaped buffers and transmit the packets at link speed, and any variance in packet transmit times constitute post-processing noise.

Limitations. Our prototype has two limitations in enforcing secret-independent timing. First, our QUIC implementation uses standard OpenSSL, which may not provide constant-time crypto. However, QUIC can be modified to adopt a constant-time crypto library [1, 2] to overcome this limitation. Secondly, it is difficult to find the true maximum values of T_{prep} and T_{enq} on general-purpose desktops. If Prepare's execution exceeds the profiled max values, it violates the theoretical DP guarantees. However, we note that it is difficult to practically exploit these violations for inferring traffic secrets.

4.3 Security Analysis

NetShaper provides the following security property: an adversary cannot infer application secrets from observing tunnel traffic. This property is ensured by a combination of a secure shaping strategy, the tunnel design, and implementation.

S1. Secure shaping strategy. The tunnel transmits traffic in differentially private-sized bursts in fixed intervals. Thus, the overall shape is DP. The proof of DP is in §C.

S2. Secure tunnel design. (i) The privacy guarantees of a tunnel are configured before the start of application transmission and do not change during the tunnel's lifetime. (ii) The tunnel mediates control between the end hosts, e.g., by transmitting custom connection establishment and termination messages. These messages are subject to the same DP shaping as the payload traffic (§4.1). (iii) The payload and dummy bytes in network packets are indistinguishable because all payload and dummy bytes are packaged into QUIC packets and encrypted uniformly. Moreover, QUIC handles acknowledgements, congestion control, and loss recovery for both payload and dummy bytes uniformly (§4.1).

S3. Secure middlebox implementation. (i) The unshaped traffic between an end host and its local middlebox is not visible to an adversary. (ii) DShaper follows the tunnel design in transmitting payload and dummy bytes. (iii) The time required for Prepare to prepare and enqueue shaped buffers is masked to secret-independent times. The packetization of buffers in QUIC is secret-independent (§4.2) and thus retains DP guarantees after post-processing. Any delays in transmitting the buffers can arise only due to congestion or packet losses in the tunnel network, which are secret-independent events.

4.4 Deployment and Maintenance

NetShaper's tunnel endpoint design and implementation are both very modular and portable. The middlebox components are compatible with all application layer protocols (e.g., HTTPS, QUIC-TLS) and network stacks (e.g., TCP, UDP, QUIC stacks). The `UShaper` could also be implemented as a standard SOCKS5 proxy [3].

A tunnel endpoint could be integrated with any node along an application's network path as long as the application traffic is unobservable until egress from the tunnel. By integrating with a trusted VPN gateway of an organization, network administrators could manage "long-term" tunnels between the organization's campuses and support multiple applications without modifying individual end hosts. For instance, separate tunnels may be configured per-application according to the organizational needs, or configurations may be adapted based on coarse-grained changes in the traffic patterns through the day.

Alternatively, by integrating with end user devices, e.g., with VPN clients, users could instantiate a new bidirectional tunnel with a service before each network activity, choose a different configuration for each tunnel instance, and close the tunnel after completion of the activity. A key requirement would be to secure the tunnel endpoint's execution from any internal side channels on the end host, which would be now more prevalent than in the middlebox setup.

5 Evaluation

Our evaluation answers the following questions. (i) How well does NetShaper mitigate state-of-the-art network side-channel attacks? (ii) What are the overheads associated with varying DP relevant configuration parameters? (iii) What are the packet latency overheads and the peak line rate and throughput sustained by our NetShaper middlebox? (iv) What are NetShaper's overall costs on privacy, bandwidth, client latency, and server throughput for different classes of applications? (v) How do NetShaper's privacy guarantees and performance overheads compare to prior techniques?

For our experiments, we use four AMD Ryzen 7 7700X desktops each with eight 4.5 GHz CPUs, 32 GB RAM, 1 TB storage, and one Marvell AQC113CS-B1-C 10Gbps NIC. We simulate client and server applications on two of the desktops and NetShaper's middleboxes on the other two desktops. The middleboxes are connected to each other via an additional Intel X550-T2 10Gbps NIC on each desktop. The client and server desktops are connected to one of the two middleboxes each via the Marvell NICs, overall forming a linear topology.

We implemented the `UShaper` and `DShaper` processes in 1100 and 1800 lines of C++ code, respectively, and deployed them on Ubuntu OS 22.04.02 (kernel version 5.19). NetShaper relies on the MSQUIC implementation of QUIC, libmsquic v2.1.8, which includes OpenSSL for traffic encryption, contributing an additional 180713 LoC to NetShaper's TCB.

For rapid evaluation, we built a simulator, which implements the `Prepare` thread's DP logic. The simulator transforms an application's original packet sequence (from tcpdump) into a sequence of burst sizes within fixed-length intervals, and outputs a sequence of transmit sizes corresponding to DP queries. We confirmed that the bandwidth overheads from the simulator closely match the overheads observed on the testbed. Thus, we report privacy and bandwidth overhead results from the simulator and latency and throughput results from the testbed, unless specified otherwise.

We use two applications for case studies, a video streaming service and a medical web service, which we describe below. Both applications are hosted on an Nginx 1.23.4 web server and the datasets are stored on the host file system.

Video service. The video streaming service is used to serve 100 YouTube videos in 720p resolution with 5 min to 130.3 min (median 12.6 min) durations and 2.7 MB to 1.4 GB (median 73.7 MB) sizes. We implement a custom video streaming client in Python, which uses one TCP connection for a single stream and requests individual 5s segments from the service synchronously. Unless specified otherwise, we stream the first 5 min of videos in our experiments. The configuration is in line with that used in prior work [55] and reasonable, given the popularity of short videos [60].

Web service. The web service is used to serve a corpus of 96 static HTML pages of a medical website, whose sizes range from 54 KB to 147 KB (median: 90 KB). As a client, we use modified wrk2 [5] that issues concurrent asynchronous HTTPS GET requests at a specified rate.

For all experiments, we use one baseline setup and one of three NetShaper configurations. In the baseline setup (**Base**), the client is directly connected to the server. In the simulator setup ($\mathbf{NS_S}$), we generated sequences of shaped burst sizes using DP shaping. With NetShaper, the traffic between the client and the server passes through two middleboxes, each implementing `UShaper` and `DShaper`. We consider two configurations of the middleboxes: (i) $\mathbf{NS_M}$: `DShaper` does not implement shaping (i.e., neither DP noise sampling nor the fixed length loop interval T_{max}), allowing us to measure the system overheads due to the middlebox implementation, and (ii) **NS**: `DShaper` implements the full shaping mechanism. By default, each middlebox is configured with 128 pairs of per-flow transmit and receive queues (unless specified otherwise). We configure the queue sizes for the max data that can be transmitted at line rate for a given DP shaping interval. We configure a max cutoff for the shaped buffer length based on the max burst length of the application and the number of active flows. This implies that the number of flows is public.

In addition, we compare with two other shaping strategies: constant-rate shaping (**CR**), which is the most secure shaping strategy, and Pacer (**Pacer**), a SOTA system that shapes traffic on a per-request basis. For **CR**, we configure the peak load corresponding to the largest object sizes in our applications, which involves transmitting 1.7MB in 5s for videos and 57KB

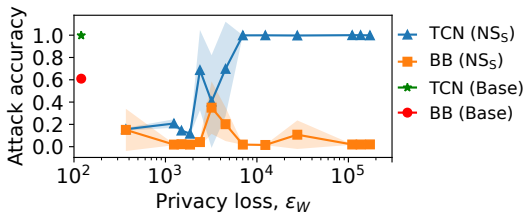
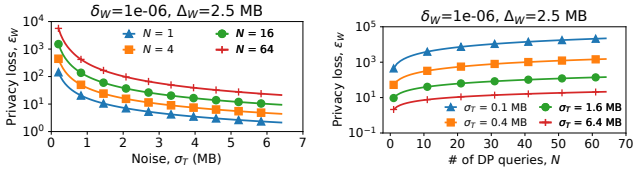


Figure 5: Classifier accuracy on shaped traces.



(a) (b)

Figure 6: Privacy loss vs (a) noise and (b) # of DP queries.

in 50ms for web. For Pacer, we pad all web pages to the largest page size, i.e., 147KB in our dataset. For videos, Pacer pads a segment at i^{th} index in a video stream to the largest segment size at that index across all videos in the dataset.

5.1 NetShaper Defeats Attack Classifiers

We start with an empirical evaluation of the privacy offered by NetShaper's traffic shaping. Recall that the traffic shaping depends on several DP parameters: the window length W, the sensitivity Δ_T, the length of the DP interval T, and the privacy loss ε. We evaluated the classifiers from §2.1 on shaped traffic generated using various values of these DP relevant parameters. We present the classifiers' performance based on only one set of values for W, Δ_T, and T, while varying ε between [200, 200000]. Our goal is to provide intuition about what values of ε are sufficient to thwart a side-channel attack.

We set (i) $W = 5s$ to align with the 5s video segments that comprise the videos, (ii) $\Delta_T = 2.5MB$, which covers 99th %ile of the distances in our dataset (§B), and (iii) $T = 1s$, which leads to composing the privacy loss over $N = 5$ DP queries on the buffering queues. The 1s interval provides a reasonable trade-off between privacy loss, and bandwidth and latency overheads, which we discuss in the subsequent sections.

We used 40 videos of 5 min duration each. We streamed each video 100 times through our testbed without shaping and collected the resulting tcpdump traces. For each value of ε, we transformed each unshaped trace into a shaped trace using our simulator to generate a total of 4000 shaped traces. We also shaped traces for **CR** and **Pacer**.

We train and test BB and TCN on shaped traces as in §2.1 and report the average and standard deviation of the accuracy of each classifier over three runs. Recall from §2.1, the BB and TCN accuracy on unshaped traffic (**Base**) is 0.61 and 0.99, respectively. For **CR** and **Pacer**, the accuracy of both BB and TCN classifiers is 0.025. This is expected since both strategies transform all unshaped traces into a single shape.

Figure 5 shows the classifiers' performance with NetShaper. While BB does not perform well for nearly all values of ε, TCN can be thwarted for ε upto 1000. *While even* $\varepsilon \approx 200$ *is too large to offer meaningful theoretical privacy guarantees, it is sufficient to defeat SOTA attacks.*

5.2 Impact of Privacy Parameters

We now evaluate how ε varies with Δ_T, σ_T, and N. Due to space constraints, we present plots for a fixed value of $\Delta_T = 2.5MB$ and defer other plots to §B. All analyses use $\delta = 10^{-6}$.

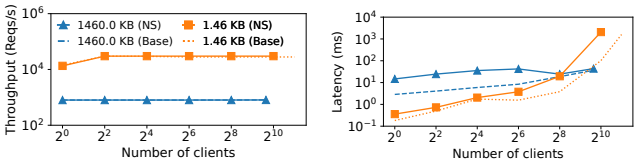
Figure 6a shows the tradeoff between ε and σ_T over four different values of N. Intuitively, a larger σ_T implies higher bandwidth overhead due to DP shaping. To retain a total privacy loss $\varepsilon = 1$ with at most 4 DP queries, we need to add noise with $\sigma_T = 18MB$ for each DP query. In contrast, $\varepsilon = 200$ with 4 DP queries (approx. configuration that defeats the classifiers in §5.1) only requires $\sigma_T < 0.3MB$. Figure 6b shows that the total privacy loss escalates with an increase in the number of DP queries. While fewer queries within a window (thus larger decision intervals) help to lower the total privacy loss, the tradeoff is the higher latency overhead. We discuss this tradeoff, as well as reduction in bandwidth overheads with concurrent flows in §5.4 and §5.5.

Using these plots, an application can choose suitable values of W and Δ_T to determine the tradeoff between ε and σ_T. For our web application serving static HTML, we recommend $W = 1s$, since web page downloads in our AWS setup (§2.1) finished within 1s, and $\Delta_T = 60KB$, which covers all distances. §5.1 explained the choices for our video application. Using ε and T, we can further determine the aggregate privacy loss over longer traffic streams using Rényi-DP composition. For instance, with $\varepsilon = 1$, $W = 5s$, and $T = 1s$, the total privacy loss for a 5 min video, which generates 300 DP queries at 1s intervals, is 8.92; the total loss for a 1 hr video is 38.8. We emphasize that Δ_T *and* ε *should be selected using plots like Figure 6, independently of the application's dataset.*

5.3 Performance Microbenchmarks

We now turn our attention to experiments to determine the overheads on per-packet latencies and the peak line rate and throughput sustainable by a NetShaper middlebox.

Middlebox throughput. We measure the peak throughput (requests/s) attained by a server application, the response latency experienced by the clients, and the impact on this throughput and latency due to the middleboxes. We restrict Nginx to one worker thread and one core on the server desktop. We evaluate using two object sizes: 1.4 KB (one MTU) and 1.4 MB. We identify the peak request rate that can be handled by a single wrk2 client, then increase the clients until we



(a) # clients vs throughput (b) # clients vs latency

Figure 7: Throughput and latency overhead due to middleboxes without shaping.

find the peak throughput the server can provide. We then vary the number of concurrent clients while generating the peak request load sustainable by the server to find the maximum number of concurrent clients that the server can handle and to measure the impact on the response latency.

We run experiments for 3 min and discard the measurements from the first minute to eliminate startup effects. Figure 7a shows the average of the peak throughput observed across 3 runs. The standard deviation is below 1% in all cases. For 1.4 KB and 1.4 MB objects, the **Base** server achieves a peak throughput of 30K req/s (64 clients) and 800 req/s (800 clients), respectively. $\mathbf{NS_M}$ matches the peak throughput and the max concurrent clients sustained by **Base**.

Latency. Figure 7b shows the average and standard deviation of the response latencies over a 2 min run. The ping latency between each pair of directly connected desktops is 0.56 ± 0.18 ms. This overhead comes from the fact that each packet traverses four additional network stacks (across two middleboxes) in each direction. This also involves data copy operations between the kernel and user space. The data copy overhead is proportional to the object size; thus $\mathbf{NS_M}$'s latency overhead increases with the larger response sizes.

The kernel data copy overheads are not fundamental to NetShaper's design. By using kernel bypass techniques or tools like DPDK [28], NetShaper can reduce the latency overhead.

Shaping interval, preparation, and enqueue times. We further profile the middlebox execution to measure the max latencies of the two components in the `Prepare` loop (§4.2): the preparation of the shaped buffer and queuing of the buffer to QUIC worker. These measures determine the maximum durations for preparing and enqueueing shaped buffers (T_{prep} and T_{enq}, respectively), and the minimum value for the shaping interval T. We profile the delays with the middlebox configured with 128 queues, thus supporting a maximum of 128 concurrent clients. One can profile the delays for a different number of queues and concurrent clients.

Based on our measurements, we set $T_{prep} = 6ms$ and $T_{enq} = 1ms$. The smallest value for T that we can configure is 10ms.

Throughput and latency with shaping enabled. We now re-run the microbenchmarks with **NS** configuration. We use three different configurations for T: 10ms, 50ms, and 100ms. We use 128 concurrent clients. The middlebox can sustain the peak throughput of 30K req/s with 1.4KB objects and 700 req/s with 1.4MB objects for each configuration of T. For 1.4KB objects, the average and standard deviation of the response latency with the three configurations are as follows: (i) $T = 10ms$: 30.47 ± 3.89 ms, (ii) $T = 50ms$: 51.39 ± 14.64 ms, (iii) $T = 100ms$: 77.49 ± 28.96 ms. For 1.4MB objects, the latencies are as follows: (i) $T = 10ms$: 41.31 ± 10.84 ms, (ii) $T = 50ms$: 76.96 ± 21.12 ms, (iii) $T = 100ms$: 127.48 ± 45.69 ms. The latency is dominated by the T configuration. The high variance in the latency is due to shaping. If a request arrives just after the decision loop has prepared a buffer in the current iteration, the request will be delayed by at least one iteration of the loop. Moreover, a negative sampling of DP noise may lead to a smaller shaped buffer than the available payload bytes in the buffering queues, thus delaying the requests by one or more intervals. This effect is particularly enhanced in a workload close to the line rate. Thus, NetShaper can perform well within about 12-15% of the line rate.

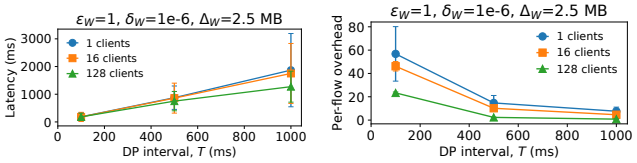
CPU utilization. The CPU utilization is 3-10% for the `Prepare` core and depends on the DP shaping interval; the utilization is 8-70% for the QUIC worker core, which depends on the network I/O. The `UShaper` core utilizes 100% of the CPU as it polls for packets from `Prepare`. As such, the `Prepare` and QUIC worker cores would be able to support additional tunnel instances by time-sharing their core. By using a polling interval, we could reduce the CPU utilization of `UShaper` to support additional requests at the cost of additional latency. In general, multiple tunnels can time-share the same physical cores, as long as each core runs the same type of thread, to suffice property P4 mentioned in §4.2.

5.4 Case Study: Video Streaming

Next, we examine the effect of different privacy settings on bandwidth and latency overheads for video streaming clients.

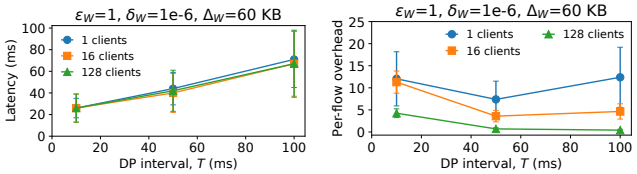
We run experiments with three values of T for the server: 100ms, 500ms, and 1s, and max per-flow DP length cutoff of 1.21 MB, 1.22MB, and 1.7 MB, respectively. We use $\Delta_T = 2.5MB$ and $\varepsilon = 1$. For all experiments, we set the DP parameters for client request traffic as follows: $\Delta_T = 200$ bytes, $W = 1s$, $T = 10ms$, $\varepsilon = 1$ and max per-flow DP length cutoff of 206 bytes. We run experiments with 1, 16, and 128 video clients; each client requests one video randomly selected from the dataset. For each set of configurations, we measure the average response latency for individual video segments with the testbed as well as the per-flow relative bandwidth overhead for the video streams in the simulator.

Latency and bandwidth overhead. Figures 8a and 8b respectively show the average segment download latency and the average per-flow relative bandwidth overhead as a function of different intervals and for varying number of clients. The **Base** segment download latency is 2.86 ± 1.41ms. The latency variance is due to variances in the segment sizes. The relative bandwidth overhead of a video is the number of dummy bytes transmitted normalized to the size of the unshaped video stream. The error bars show the standard



(a) Latency vs DP interval

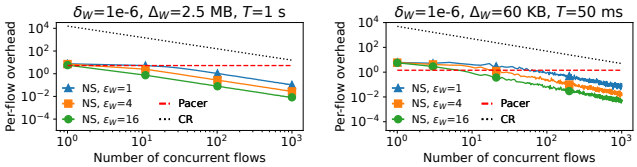
(b) BW overhead vs DP interval



(c) Latency vs DP interval

(d) BW overhead vs DP interval

Figure 8: Latency and bandwidth overhead for different values of DP interval for video streaming (a, b) and for web (c, d).



(a) Video streaming.

(b) Web service.

Figure 9: Comparison of NetShaper, constant shaping, Pacer.

deviation in latency and bandwidth overhead.

First, Figure 8a shows that, for all values of T, the video segments can be downloaded well within 5s, which is the time to play each segment and request the next segment from the server. The high variance is due to negative DP noise, which delays payload transmission. Secondly, the results show the trade-off between latency and bandwidth. A larger DP interval implies higher download latency but fewer queries, thus yielding a lower bandwidth overhead. Thirdly, with multiple concurrent clients, the bandwidth overhead is amortized, while the average download latency only depends on T. Overall, NetShaper can secure video streams with low bandwidth overheads and no impact on the streaming experience.

5.5 Case Study: Web Service

We perform similar experiments as §5.4 with our web service. For the server responses, we use $\Delta_T = 60KB$, $W = 1s$, and $\varepsilon = 1$. We use T of 10ms, 50ms, and 100ms, and per-flow DP length cutoffs of 60.8KB, 60.8KB, and 110.9KB, respectively. For the client requests, we use the same configs as in §5.4. We run 3 min experiments with 1, 16, and 128 wrk2 clients with a total load of 1600 req/s; each client requests random web pages from the dataset. We discard the numbers of the first minute.

Latency and bandwidth overhead. Figures 8c and 8d respectively show the average response latency and the average per-web page relative bandwidth overhead, across all web page requests. The **Base** latency is 0.225 ± 0.3 ms. (The high variance is due to the time precision in wrk2 being restricted to 1ms.) The web workload is more sporadic than video streaming, thus web page download latencies have higher variance than video segment download latency. The absolute latency overhead for **NS** depends on the choice of T. The relative overhead depends on the underlying network latency, which unlike our testbed, is in the order of 10s to 100s of milliseconds in the Internet. Interestingly, the bandwidth overhead for web traffic first reduces with increasing DP shaping interval from 10ms to 50ms, but then increases again with an interval of 100ms. This is because, for small web pages, the DP interval of 100ms is larger than the total time required to download web pages. As a result, additional overhead is incurred due to the padding of traffic in the 100ms intervals.

5.6 Comparison with other techniques

Figures 9a and 9b show the per-flow relative bandwidth overhead of **NS**, **CR**, and **Pacer** for video and web applications, respectively, for varying number of concurrent flows.

For both video and web traffic, **NS** incurs three orders of magnitude lower overhead than **CR**, which requires continuously transmitting traffic at the peak server load (configured for 1000 clients). For video and web traffic, **NS** requires 11 flows and more than 40 flows, respectively to achieve lower overhead **Pacer**. Pacer shapes server traffic only upon receiving a client request and does not shape client traffic. Thus, it leaks the timing and shape of client requests, which could potentially reveal information about the server responses [19]. NetShaper shapes traffic in both directions, which incurs higher overhead at the cost of stronger privacy than Pacer.

Evaluation summary. Our evaluation provides four insights. (i) There is a huge gap between the theoretical DP guarantees and the privacy configurations required to defeat SOTA attacks. (ii) The latency overhead is dominated by the choice of DP shaping interval, (iii) NetShaper's middlebox can match about 88% of the 10Gbps NIC line rate; a single core of `UShaper` can match the peak throughput of a single core server, (iv) NetShaper's cost is in the two additional cores for `Prepare` and QUIC worker, which helps to avoid any secret-dependent interference in shaping and keep low DP shaping loop lengths. By optimising the implementation, we could use a single middlebox to support larger workloads.

6 Related Work

Traffic shaping for web. Prior work used traffic shaping for defending against website fingerprinting attacks. Walkie-Talkie [67], Supersequence [66], and Glove [50] use clustering techniques to group objects of a corpus and then shape the traffic of all objects within each cluster to conform to a similar pattern. Traffic morphing [71] makes the traffic of one page look like that of another. These techniques compute traffic

shapes that envelope or resemble the network traces of individual objects. Hence, they require a large number of traces to account for network variations. NetShaper dynamically adapts traffic shapes based on the prevailing network conditions.

Cs-BuFLO [16], Tamaraw [17], and DynaFlow [43], determine traffic shape directly at runtime. They pad object sizes to values that are correlated with the original object sizes, such as the next multiple or power of a configurable constant. These defenses provide privacy akin to k-anonymity, but have no control on the the size of the anonymous cluster. Tamaraw [17] formalizes the privacy guarantees. NetShaper's DP guarantees are strictly stronger than Tamaraw's (proof in §D).

Differential privacy over streams. Dwork et al. [24] study DP on streams, in which neighboring streams differ in at most one element and the query counts over the stream prefix (without forgetting old information) under a fixed DP budget for the entire stream. NetShaper requires a stronger neighboring definition to model application data streams (Def. 1), but is able to forget the past by dropping stale data from our queue (Assumption 1) and compose privacy loss over time.

NetShaper's neighboring definition is closer to that of user-level DP over streams in [25]. Instead of counting discrete change, however, we use the L1 distance which enables coarsening. Pan-privacy considers an adversary that can compromise the internals of the algorithms (e.g., our buffering queue). This makes the design of algorithms challenging and costly. Instead, we consider the buffering queue private and focus on practical algorithms to study the privacy/overheads trade-off.

Kellaris et al. [38] introduce a notion of DP, called w-event privacy, for streams of length w. Neighboring stream pairs are those whose individual events are pairwise neighbors within a window of upto w. NetShaper's neighboring definition accounts for the maximum stream distance over *any* window of length W, which is a better match for the streams we consider.

Zhang et al. [73] generate differentially private shapes for video streams using Fourier Perturbation Algorithm (FPA) [52]. FPA transforms a finite time series of bursts, into a series of DP shaped bursts of the same length. FPA requires the entire stream's profile upfront, and cannot guarantee complete transmission of an input stream within the shaped trace. NetShaper's DP guarantees simply compose over burst interval sequences, thus allowing shaping of streams of arbitrary lengths with quantifiable privacy and overheads.

Adversarial defenses. Adversarial defenses [6, 30, 34, 49, 51, 56] generate targeted and low-overhead noise to defeat specific classifiers. NetShaper provides provable and configurable privacy against both SOTA as well as future classifiers.

Network side-channel mitigation systems. NetShaper's shaping tunnel is conceptually similar to Pacer's [45] cloaked tunnel. However, Pacer mitigates leaks of a Cloud tenant's secrets to a colocated adversary through contention at shared network links. Pacer's cloaked tunnel controls the transmit time of TCP packets in accordance with the shaping schedule and congestion control signals. Thus, Pacer requires non-trivial changes to the network stack on the end hosts. NetShaper protects applications that are behind private networks but communicate using the public Internet. NetShaper's tunnel endpoints can be placed at the interface of the private-public networks, e.g., in a middlebox, thus supporting multiple applications without modifying end hosts. Moreover, by shaping above the transport layer, NetShaper needs to control only the precise timing for generation of bursts of DP length and not the subsequent transmission to the network.

Ditto [46] and NetShaper propose shaping traffic at network nodes separate from end hosts. NetShaper proposes a hardware-independent, modular and portable middlebox architecture that can be integrated with end hosts, routers, gateways, or even programmable switches as in Ditto.

Systems with other goals. Censorship circumvention systems [10, 11, 48, 53, 70] rely on traffic obfuscation, scrambling, and transformations of a sensitive application's shape to that of a non-sensitive application. These techniques prevent identification of original protocols by deep packet inspection, but do not prevent inference of secrets from traffic shapes.

Karaoke [40] and Vuvuzela [62] are anonymous messaging systems that use DP to hide participants in a conversation, but use constant-rate traffic among the participants. AnoA [8] is a framework to analyze anonymity properties of anonymous communication protocols. AnoA supports DP based quantification for various properties, such as sender anonymity and sender unlinkability. NetShaper's differentially-private traffic shaping hides the traffic content. In principle, NetShaper could be combined with an anonymous communication system to provide both content privacy and anonymity with DP.

7 Conclusion

NetShaper is a provably secure network side-channel mitigation system that provides quantifiable and tunable privacy guarantees in traffic shaping. We believe that NetShaper can eliminate the arms race in network side-channel attacks and defenses and can provide a portable and configurable framework for deploying mitigations for applications with diverse traffic characteristics and in different settings. NetShaper's DP based traffic shaping strategy as well as its modular and portable tunnel design can be extended to mitigate leaks in multi-node systems, but we leave the details to future work.

8 Acknowledgments

We thank the reviewers and our shepherd for their constructive feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2021-02961, DGDND-2021-02961], the Innovation for Defence Excellence and Security (IDEaS) Program of the Department of National Defense [MN3-011], a Google Research Scholar award, the Digital Research Alliance of Canada, and AWS (through UBC Cloud Innovation Center).

References

[1] A Highw Assurance Cryptographic Library. https://hacl-star.github.io/. Accessed on Sep 20, 2023.

[2] libsecp256k1. https://github.com/bitcoin-core/secp256k1. Accessed on Sep 20, 2023.

[3] Pluggable Transports. https://obfuscation.github.io/. Accessed on Jun 6, 2023.

[4] What is TCP Meltdown? https://openvpn.net/faq/what-is-tcp-meltdown/. Accessed on Apr 30, 2023.

[5] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[6] A. Abusnaina, R. Jang, A. Khormali, D. Nyang, and D. Mohaisen. DFD: Adversarial Learning-based Approach to Defend Against Website Fingerprinting. *IEEE INFOCOMM*, 2020.

[7] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying Constant-Time Implementations. *USENIX Security*, 2016.

[8] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. AnoA: A Framework for Analyzing Anonymous Communication Protocols. *IEEE CSF*, 2013.

[9] S. Bai, J. Z. Kolter, and V. Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *arXiv:1803.01271*, 2018.

[10] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes. Poking a Hole in the Wall: Efficient Censorship-Resistant Internet Communications by Parasitizing on WebRTC. *ACM CCS*, 2020.

[11] D. Barradas, N. Santos, and L. E. Rodrigues. DeltaShaper: Enabling Unobservable Censorship-resistant TCP Tunneling over Videoconferencing Streams. *PETS*, 2017.

[12] A. Beams, S. Kannan, and S. Angel. Packet Scheduling with Optional Client Privacy. *ACM CCS*, 2021.

[13] M. Beckerle, J. Magnusson, and T. Pulls. Splitting Hairs and Network Traces: Improved Attacks Against Traffic Splitting as a Website Fingerprinting Defense. *WPES*, 2022.

[14] S. Bhat, D. Lu, A. Kwon, and S. Devadas. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning. *PETS*, 2019.

[15] B. A. Braun, S. Jana, and D. Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. *arXiv:1506.00189*, 2015.

[16] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. *WPES*, 2014.

[17] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. *ACM CCS*, 2014.

[18] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the Scope of Differential Privacy Using Metrics. *PETS*, 2013.

[19] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. *IEEE S&P*, 2010.

[20] G. Cherubin, J. Hayes, and M. Juarez. Website Fingerprinting Defenses at the Application Layer. *PETS*, 2017.

[21] G. Danezis. Traffic Analysis of the HTTP Protocol over TLS. http://www0.cs.ucl.ac.uk/staff/G.Danezis/papers/TLSanon.pdf, 2009.

[22] W. De la Cadena, A. Mitseva, J. Hiller, J. Pennekamp, S. Reuter, J. Filter, T. Engel, K. Wehrle, and A. Panchenko. TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting. *ACM CCS*, 2020.

[23] J. Dong, A. Roth, and W. J. Su. Gaussian Differential Privacy. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 2022.

[24] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential Privacy under Continual Observation. *ACM STOC*, 2010.

[25] C. Dwork, M. Naor, T. Pitassi, G. N. Rothblum, and S. Yekhanin. Pan-Private Streaming Algorithms. *ACM ICS*, 2010.

[26] C. Dwork, A. Roth, et al. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4), 2014.

[27] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. *IEEE S&P*, 2012.

[28] L. Foundation. Data plane development kit (DPDK), 2015.

[29] J. Gong and T. Wang. Zero-delay Lightweight Defenses against Website Fingerprinting. *USENIX Security*, 2020.

[30] J. Gong, W. Zhang, C. Zhang, and T. Wang. Surakav: Generating Realistic Traces for a Strong Website Fingerprinting Defense. *IEEE S&P*, 2022.

[31] J. Hayes and G. Danezis. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. *USENIX Security*, 2016.

[32] N. P. Hoang, A. A. Niaki, P. Gill, and M. Polychronakis. Domain Name Encryption is Not Enough: Privacy Leakage via IP-based Website Fingerprinting. *PETS*, 2021.

[33] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama. Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency. *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, 2005.

[34] C. Hou, G. Gou, J. Shi, P. Fu, and G. Xiong. WF-GAN: Fighting Back Against Website Fingerprinting Attack Using Adversarial Learning. *IEEE ISCC*, 2020.

[35] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[36] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. Toward an Efficient Website Fingerprinting Defense. *ESORICS*, 2016.

[37] S. P. Kasiviswanathan and A. Smith. On the 'Semantics' of Differential Privacy: A Bayesian Formulation. *Journal of Privacy and Confidentiality*, 2014.

[38] G. Kellaris, S. Papadopoulos, X. Xiao, and D. Papadias. Differentially Private Event Sequences over Infinite Streams. *VLDB Endowment*, 7(12), 2014.

[39] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. *USENIX Security*, 2012.

[40] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. *USENIX OSDI*, 2018.

[41] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified Robustness to Adversarial Examples with Differential Privacy. *IEEE S&P*, 2019.

[42] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. *IEEE HPCA*, 2016.

[43] D. Lu, S. Bhat, A. Kwon, and S. Devadas. DynaFlow: An Efficient Website Fingerprinting Defense Based on Dynamically-Adjusting Flows. *WPES*, 2018.

[44] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. *NDSS*, 2011.

[45] A. Mehta, M. Alzayat, R. De Viti, B. B. Brandenburg, P. Druschel, and D. Garg. Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud. *USENIX Security*, 2022.

[46] R. Meier, V. Lenders, and L. Vanbever. ditto: WAN Traffic Obfuscation at Line Rate. *NDSS*, 2022.

[47] I. Mironov. Rényi Differential Privacy. *IEEE CSF*, 2017.

[48] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. Skypemorph: Protocol Obfuscation for Tor Bridges. *ACM CCS*, 2012.

[49] M. Nasr, A. Bahramali, and A. Houmansadr. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. *USENIX Security*, 2021.

[50] R. Nithyanand, X. Cai, and R. Johnson. Glove: A Bespoke Website Fingerprinting Defense. *WPES*, 2014.

[51] M. S. Rahman, M. Imani, N. Mathews, and M. Wright. Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks with Adversarial Traces. *IEEE Trans. on Information Forensics and Security*, 16, 2020.

[52] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. *ACM Intl. Conf. on Management of Data*, 2010.

[53] M. B. Rosen, J. Parker, and A. J. Malozemoff. Balboa: Bobbing and Weaving around Network Censorship. *USENIX Security*, 2021.

[54] T. S. Saponas, J. Lester, C. Hartung, S. Agarwal, T. Kohno, et al. Devices That Tell on You: Privacy Trends in Consumer Ubiquitous Computing. *USENIX Security*, 2007.

[55] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. *USENIX Security*, 2017.

[56] S. Shan, A. N. Bhagoji, H. Zheng, and B. Y. Zhao. Patch-based Defenses against Web Fingerprinting Attacks. *AISec*, 2021.

[57] T. Shapira and Y. Shavitt. Flowpic: Encrypted Internet Traffic Classification is as Easy as Image Recognition. *IEEE INFOCOMM Workshops*, 2019.

[58] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. *ACM CCS*, 2018.

[59] J.-P. Smith, L. Dolfi, P. Mittal, and A. Perrig. QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework. *USENIX Security*, 2022.

[60] Statista. Length of online videos watched on social media platforms worldwide in August 2021, by age group. https://tinyurl.com/9u4ystpe, 2023. Accessed on Sep 20, 2023.

[61] G. Tan. Principles and Implementation Techniques of Software-Based Fault Isolation. *Foundations and Trends® in Privacy and Security*, 1(3), 2017.

[62] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. *ACM SOSP*, 2015.

[63] V. Varadarajan, T. Ristenpart, and M. M. Swift. Scheduler-based Defenses against Cross-VM Side-channels. *USENIX Security,* 2014.

[64] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. *ACM EuroSys*, 2012.

[65] M. Wang, A. Kulshrestha, L. Wang, and P. Mittal. Leveraging Strategic Connection Migration-powered Traffic Splitting for Privacy. *Privacy Enhancing Technologies (PETS)*, 2022.

[66] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. *USENIX Security*, 2014.

[67] T. Wang and I. Goldberg. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. *USENIX Security*, 2017.

[68] C. Westphal, S. Lederer, C. Mueller, A. Detti, D. Corujo, J. Wang, M.-J. Montpetit, N. Murray, C. Timmerer, D. Posch, A. Azgin, and W. Liu. RFC 7933: Adaptive Video Streaming over Information-Centric Networking (ICN). https://datatracker.ietf.org/doc/html/rfc3168. Accessed on Apr 30, 2023.

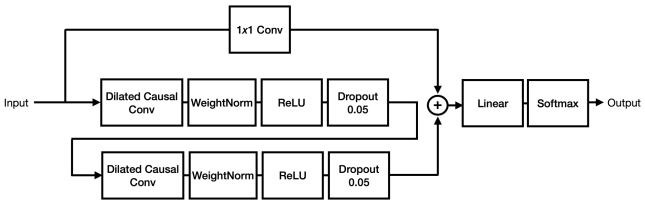


Figure 10: TCN classifier

[69] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose. Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Fon-iks. *IEEE S&P*, 2011.

[70] P. Winter, T. Pulls, and J. Fuss. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. *WPES*, 2013.

[71] C. V. Wright, S. E. Coull, and F. Monrose. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. *NDSS*, 2009.

[72] D. Zhang, A. Askarov, and A. C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. *ACM CCS*, 2011.

[73] X. Zhang, J. Hamm, M. K. Reiter, and Y. Zhang. Statistical Privacy for Streaming Traffic. *NDSS*, 2019.

A Attack classifiers

Beauty and Burst. The Beauty and the Burst classifier (BB) [55] is a CNN (convolutional neural network) consisting of three convolution layers, a max pooling layer, and two dense layers. We use a dropout of 0.5, 0.7, and 0.5 between the hidden layers of the network. We train the classifier with an Adam optimizer, a categorical cross-entropy function, a learning rate of 0.01, with a batch size of 64, and for 1000 epochs.

Temporal Convolution Network. While CNNs are generally effective in sequence modelling, they look at future samples in a sequence and a very limited history of past samples to decide the output of the current sample. Consequently, they require a large number of traces and long traces for effective training and prediction.

Temporal Convolutional Networks (TCNs) [9] overcome these problems of CNNs by utilizing a one-dimensional fully-convolutional network equipped with causal dilated convolutions, which allows them to examine deep into the past to produce an output for the sequence at any given moment.

Figure 10 shows the architecture of our TCN classifier, which follows the architecture proposed by Bai et al. [9]. It consists of two dilated causal convolutional layers, followed by weight normalization and dropout layers with a dropout probability of 0.05. We train the classifier for 1000 epochs.

B Extended evaluation of privacy vs overheads

Distribution of queue length differences. Figure 11 shows the distribution of the buffering queue length differences generated for each pair of an application's streams. We use W of 5s for video streams and 1s for web pages. The dashed lines show the median, which is 1.63 MB and 6 KB for videos and web pages, respectively.

Privacy loss vs overheads for other Δ_T. Figure 12 shows similar results as Figure 6 for Δ_T of 0.1 MB and 10 MB.

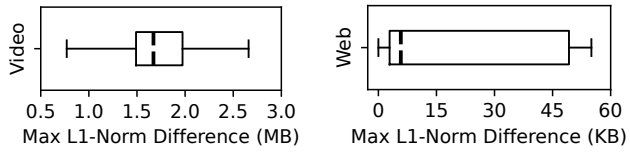
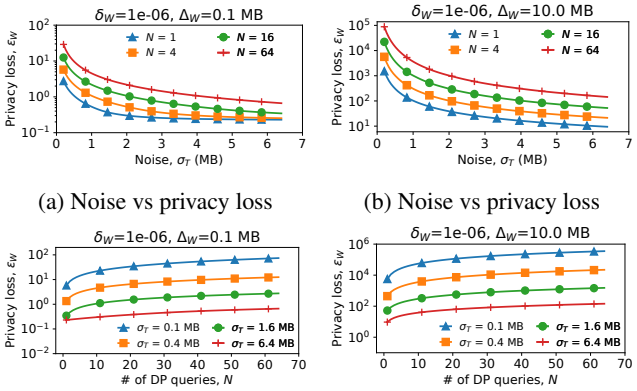


Figure 11: Distribution of the difference in buffering queue lengths for application stream pairs.



(a) Noise vs privacy loss (b) Noise vs privacy loss

(c) # DP queries vs privacy loss (d) # DP queries vs privacy loss

Figure 12: Privacy loss vs noise and # queries for different Δ_T.

Bandwidth overheads with a fixed cutoff. In §5, we explained that NetShaper applies a cutoff to the DP shaped buffer length (if the sampled noise is very large) that depends on the number of active flows. This reveals the number of active flows at any given time. To hide the number of flows, we can set the cutoff to a fixed value (e.g., based on the maximum flows that the system must support).

Figure 13a presents results similar to those of Figure 8b, but with the cutoff for the shaped buffer length set to a fixed value of 1.21 GB, 1.22 GB, and 1.7 GB. Figure 13b presents results similar to those of Figure 8d but with cutoffs of 60.8 MB, 60.8 MB, and 110.9 MB. Figures 13c and 13d show the results similar to those of Figures 9a and 9b. The fixed cutoffs correspond to 1000 flows, which lead to a significant increase in NetShaper's overheads. However, the overheads quickly amortize with several concurrent flows.

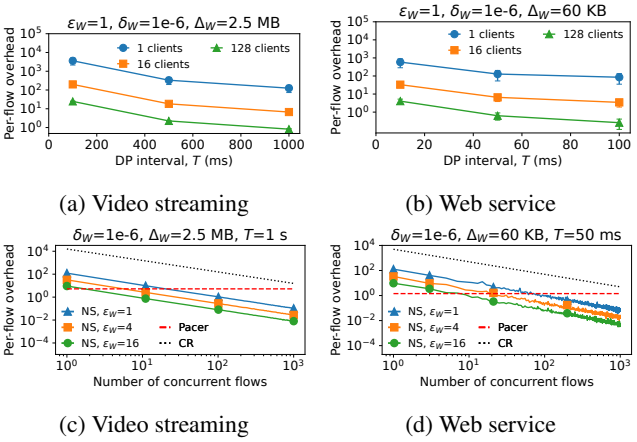
C Proof of NetShaper's DP based Shaping

Proposition 1. *NetShaper enforces* $\Delta_T \leq \Delta_W$.

To prove Prop 1, we show that at any DP query time k, any neighboring streams S, S' can only create a queue size difference such that $|L_k - L'_k| \leq \Delta_W$. This is formalized in the following Lemma:

Lemma 1. *Assume two neighboring traffic streams S and* S' ($\|S - S'\|_1 \leq \Delta_W$)*, are transmitted through NetShaper and get the same randomness draws* z_k *(they are parallel worlds with an identical NetShaper, but the streams differ). Then, at any DP query time k, the length of the buffering queue for S and* S' *are* Δ_W*-close. Formally:*

$$\forall k \geq 0 : |L_k - L'_k| \leq \Delta_W \tag{4}$$



(a) Video streaming (b) Web service

(c) Video streaming (d) Web service

Figure 13: Bandwidth overheads with a fixed cutoff.

Proof. NetShaper performs a DP query for the size of the buffering queue at intervals of length T. While transmitting S, the queue length at the end of the k^{th} interval T_k is a function of three variables: (i) The buffering queue length L_{k-1} at the end of T_{k-1}. (ii) The total number of payload bytes that have been dequeued from the buffering queue in the k^{th} interval, R_k. (iii) The number of new payload bytes from the application stream added to the buffering queue since the previous interval, which is the sum of sizes of all packets arriving between $(k-1)^{\text{th}}$ and k^{th} interval, i.e., $\sum_{T_{k-1} \leq t < T_k} P_t^S$. Therefore, the buffering queue length after dequeue in T_k is:

$$L_k = L_{k-1} + \sum_{T_{k-1} \leq t < T_k} P_t^S - R_k \tag{5}$$

The difference between the queue lengths of S, S' at T_k is:

$$L_k - L'_k = (L_{k-1} - L'_{k-1}) + \Big(\sum_{T_{k-1} \leq t < T_k} P_t^S - \sum_{T_{k-1} \leq t < T_k} P_t^{S'}\Big) - (R_k - R'_k) \tag{6}$$

We divide the proof into two steps. First, we show that the dequeue stage of the shaping mechanism does not increase the difference in queue lengths. Secondly, we show that under Assumption 1, the enqueue stage of incoming streams does not increase the difference in queue lengths beyond Δ_W.

To show that the dequeue stage never increases the difference between queue lengths, we show that in any period k, we always dequeue more (or equal) data from the larger queue at the time of the DP query. Formally:

$$(L_{k-1} - L'_{k-1}) \cdot (R_k - R'_k) \geq 0, \tag{7}$$

where the indexes are because removal amount in period k depends on the query result for the queue at $k-1$. Assume without loss of generality that $L_{k-1} \geq L'_{k-1}$. Since each DP query receives the same noise, we get $\tilde{L}_{k-1} \geq \tilde{L}'_{k-1}$, and thus:

$$R_k = \min\{0, \tilde{L}_{k-1}, L_{k-1}\} \geq \min\{0, \tilde{L}'_{k-1}, L'_{k-1}\} = R'_k.$$

The case for $L_{k-1} \leq L'_{k-1}$ is symmetric, and we have $\text{sign}(L_k - L'_k) = \text{sign}(R_k - R'_k)$, which implies Equation 7. Moreover, we show:

$$|R_k - R'_k| \leq |L_k - L'_k|, \tag{8}$$

using two cases and assuming that $L_k \geq L'_k$ (again the opposite case is symmetric). Either the DP noise is $\geq -L'_k$, and $L_k - L'_k = R_k - R'_k$.

Or the DP noise is $< -L'_k$, in which case $R'_k = 0$ but $R_k \leq L_k - L'_k$. Either way, Equation 8 holds. We can now study the queue length difference over time:

$$\begin{aligned}|L_k - L'_k| &= |(L_{k-1} - L'_{k-1}) + (\sum_{T_{k-1} \leq t < T_k} P_t^S - P_t^{S'}) - (R_k - R'_k)| \\ &\leq |(L_{k-1} - L'_{k-1}) - (R_k - R'_k)| + \sum_{T_{k-1} \leq t < T_k} |P_t^S - P_t^{S'}| \\ &\leq |L_{k-1} - L'_{k-1}| + \sum_{T_{k-1} \leq t < T_k} |P_t^S - P_t^{S'}|\end{aligned}$$

where the first equality uses Equation 6 and the last inequality uses Equation 7 and Equation 8. Intuitively, the dequeue stage never increases the difference between queue lengths.

Finally, we are ready to bound the difference in queue length due to enqueues. Let $d_k = |L_k - L'_k|$, and $d_0 = 0$:

$$\begin{aligned}d_k &\leq d_{k-1} + \sum_{T_{k-1} \leq t < T_k} |P_t^S - P_t^{S'}| = 0 + \sum_{i=0}^{k} \left(\sum_{T_{i-1} \leq t < T_i} |P_t^S - P_t^{S'}| \right) \\ &= \sum_{0 \leq t < T_k} |P_t^S - P_t^{S'}| = \sum_{0 \leq t < T_k - W} |P_t^S - P_t^{S'}| + \sum_{T_k - W \leq t < T_k} |P_t^S - P_t^{S'}| \\ &\leq \Delta_W\end{aligned}$$

since $\sum_{0 \leq t < T_k - W} |P_t^S - P_t^{S'}| = 0$ by Assumption 1, and $\sum_{T_k - W \leq t < T_k} |P_t^S - P_t^{S'}| \leq \Delta_W$ by Def 1. □

Lemma 1 $\implies$ Prop 1, since $\Delta_T = \max_k \max_{S,S'} |L_k - L'_k| \leq \Delta_W$.

D Comparison of NetShaper and Tamaraw

Tamaraw [17] provides a mathematical notion of privacy guarantee of a shaping strategy, called ε-security. To disambiguate with NetShaper's notion of (ε, δ)-DP, we rename Tamaraw's ε variable with γ. We show that NetShaper's (ε, δ)-DP definition is strictly stronger than Tamaraw's γ-security definition.

First, we explain Tamaraw's γ-security definition. In Tamaraw, W is a random variable representing the label of a traffic trace. For a trace w, the random variables T_w and T_w^D respectively represent the packet trace of w before and after shaping with a defense D. The distribution of T_w^D captures all variations in the observed shape of w resulting from both the defense mechanism and the network, while the distribution of T_w only contains variations due to the network. The attacker can measure the distribution of W and T_w^D independently. Upon observing a network trace t, an optimal attack A, selects the label of the trace with maximum likelihood of observation:

$$A(t) = \operatorname*{argmax}_w \Pr[W = w] \Pr[T_w^D = t]$$

The probability that an attack A outputs label w_i is $\Pr_A[w_i]$.

Definition (Tamaraw γ-privacy)**.** *A fingerprinting defense D is said to be uniformly γ-private if for the attack $\mathcal{A}$ if we have:*

$$\max_w \left[\Pr[A(T_w^D) = w] \right] \leq \gamma$$

Proposition 3. *Tamaraw γ-privacy is strictly weaker than the notion of ε-differential privacy.*

To prove Prop 3, we prove the following two lemmas.

Lemma 2. *There exists a Tamaraw γ-private defense mechanism that fails to satisfy ε-DP for any given value of ε.*

Proof. Consider a web service with a dataset of n web pages. We propose a defense D, which shapes pages to a constant-rate pattern O_c with probabilities α or β. D reshapes page w_j to O_c with probability β and all other pages $w_i \neq w_j$ with probability α. If a page is not shaped, it is revealed.

The probability that any attack can correctly identify the label for w_j is upper-bounded by: $\Pr[A(T^D_{w_{i=j}}) = w_j]$

$$\begin{aligned}&= \Pr[A(T^D_{w_{i=j}}) = w_j | T^D_{w_{i=j}} = T_{w_{i=j}}] \Pr[T^D_{w_{i=j}} = T_{w_{i=j}}] + \\ &\quad \Pr[A(T^D_{w_{i=j}}) = w_j | T^D_{w_{i=j}} = O_c] \Pr[T^D_{w_{i=j}} = O_c] \\ &\leq 1.(1-\beta) + \frac{1}{n}\beta = p_c^j\end{aligned}$$

For $(1 - \frac{n\gamma - 1}{n-1}) < \beta$ we have: $p_c^j \leq \gamma$. Similarly, the probability that any attack can correctly classify $w_{i \neq j}$ is upper-bounded by $p_c^i = 1 - \alpha + \frac{\alpha}{n}$, and for $(1 - \frac{n\gamma-1}{n-1}) < \alpha$ we have: $p_c^j \leq \gamma$. Therefore, for all values of α and β such that $(1 - \frac{n\gamma-1}{n-1}) < \beta < \alpha$, the probability that any attack can successfully guess a page is less than γ, and the defense is uniformly γ-private.

When the output of the algorithm is O_c, the probability of the page being w_j is β and any other page is α. Thus, $\log(\frac{\Pr[T^D_{w_{i \neq j}} = O_c]}{\Pr[T^D_{w_{i=j}} = O_c]}) = \log(\frac{\alpha}{\beta})$. By setting $\alpha > e^{\varepsilon}\beta$, we get a mechanism that is γ-private for Tamaraw but not ε-DP for NetShaper. □

Lemma 3. *A ε-DP shaping algorithm is Tamaraw γ-private for $\varepsilon \leq \log(n\gamma)$.*

Proof. For a trace w, the random variable T_w^{DP} represents the packet trace of w after a differentially private shaping mechanism is applied. The classification attack on shaped traffic can be considered a post-processing of the results of a differentially private shaping mechanism (i.e. defense) and, hence, is differentially private. Therefore, we have:

$$\frac{\Pr[A(T_{w_i}^{DP}) = w_i]}{\Pr[A(T_{w_j}^{DP}) = w_i]} \leq e^{\varepsilon} \Rightarrow \Pr[A(T_{w_i}^{DP}) = w_i] \leq e^{\varepsilon}.\Pr[A(T_{w_j}^{DP}) = w_i]$$

Intuitively, this implies that the likelihood of the attacker correctly classifying the trace with label i compared to incorrectly classifying it with label j is bounded by e^{ε}. The above inequality is correct for all $w_j : j \in \{1, 2, \ldots, n\}$, and we can extend the above equation to calculate the summation over j:

$$n \times \Pr[A(T_{w_i}^{DP}) = w_i] \leq e^{\varepsilon} \sum_{j=1}^{n} \Pr[A(T_{w_j}^{DP}) = w_i] = e^{\varepsilon} \Pr_A[w_i].$$

Thus, for any given trace w_i, the probability that any attack A, classifies it correctly is bounded by: $\Pr[A(T_{w_i}^{DP}) = w_i] \leq \frac{e^{\varepsilon} \Pr_A[w_i]}{n}$. The probability that an attacker can guess the victim's trace is bounded by: $\max_{w_i} \Pr[A(T_{w_i}^{DP}) = w_i] \leq \frac{e^{\varepsilon}}{n} \max_{w_i} \Pr_A[w_i] \leq \frac{e^{\varepsilon}}{n} \leq \gamma$. □

E Artifact Appendix

E.1 Abstract

In this paper, we present NetShaper, a system that mitigates network side channel leaks through traffic shaping. NetShaper's traffic shaping offers differential privacy guarantees while configuring a trade-off between privacy assurances, bandwidth, and latency overheads. For evaluation, we implement NetShaper on four interconnected machines, where two of these machines serve as communicating parties, with the other two functioning as middleboxes. Furthermore, to expedite the evaluation of experiments involving multiple transmissions of traffic traces, we construct a traffic shaping simulator.

E.2 Description & Requirements

E.2.1 Security, privacy, and ethical concerns

Our datasets are public. The artifact does not involve destructive actions, and there is no security, privacy, or ethical concerns associated with using it.

E.2.2 How to access

The code for both the traffic shaping simulator and our implementation of NetShaper can be found on our GitHub repository: https://github.com/ubc-systopia/netshaper/tree/AE_v2.0 . First, clone the github repository.

```
$ git clone --recursive-submodules
↪  https://github.com/ubc-systopia/netshaper.git
$ cd netshaper
```

To download NetShaper's dataset, change the directory to dataset directory and run the following commands:

```
$ cd dataset
$ wget
↪  https://zenodo.org/api/records/10783814/files-archive
$ unzip files-archive
$ tar -xf netshaper_dataset.tar.gz
```

Please note that the dataset is essential for the execution of both the simulator and the testbed implementation. Further details are available in the `README.md`.

E.2.3 Hardware dependencies

The simulator requires a machine that should have at least 8 CPU cores, 64 GB of RAM, and a GPU with 24 GB of memory. To store the dataset and experiment results, simulator needs 100 GB of disk space. For NetShaper implementation, we use four machines interconnected in a linear topology. Each machine should have at least 8 CPU cores, 32 GB of RAM, 100 GB of disk space, and a 10 Gbps NIC. The machines should be connected to a LAN, ensuring no interference from the public network or other network applications.

E.2.4 Software dependencies

Simulator requires `Python 3.10.6` and can be executed on arbitrary OS without root access. The list of required Python packages is available in our repository. To compile our code for NetShaper implementation, we use `gcc 11.4.0`. The implementation is specific to Ubuntu 22.04. For video server, we use Nginx 1.23.4. For transport protocol we use Microsoft implementation of the QUIC protocol, MsQUIC 2.2.4. We use our modified version of wrk2, an HTTP benchmarking tool, to send requests asynchronously. The list of main software dependencies are provided in Git repository.

E.2.5 Benchmarks

- **Datasets:** We use two datasets, one for our video streaming service and another for our web service. Instructions for accessing these datasets can be found in Section A.2.2.
- **Models:** We use TCN model and CNN model from the Beauty and the Burst paper. Both are included in the code repository.
- **Metrics:** We report throughput as number of requests per second, latency in milliseconds, and accuracy of the classification. The relative bandwidth overhead is the number of dummy bytes transmitted normalized to the size of the unshaped traffic trace. For privacy, we report the privacy loss, ε, as defined in the framework of differential privacy (eq. 1).
- **Data licenses:** The Creative Commons Attribution license.

E.3 Set-up

Here, we offer high-level steps for setting up both the simulator and our implementation. Additionally, each experiment is accompanied by dedicated setup instructions outlined in the corresponding section of the `README.md` file.

E.3.1 NetShaper Implementation

To set up the server and client, ensure that Docker is installed on the hosting machines, clone the `sys` repository onto both of them, download and place the dataset in the `/dataset` directory, and execute the `./setup` script for the server and the client. Our scripts will handle the remaining setup procedures. To configure the middleboxes, physical access to the machines is necessary to adjust the BIOS settings and initiate a reboot. In addition to Docker, ensure that `tcpdump` is installed on these machines to facilitate the collection of traffic traces. For more details, consult the experiments' `README.md` file.

E.3.2 Traffic Shaping Simulator

Setting up the simulator is comparatively straightforward compared to the testbed; you only need to install the necessary Python prerequisites.

```
pip install -r <path-to-simulator>/requirements.txt
```

E.3.3 Basic Test

After downloading the dataset, to test the simulator, you can run the following commands:

```
cd evaluation/web_bandwidth
./run.sh --experiment="dp_interval_vs_overhead_web"
↪ --config_file="configs/dp_interval_vs_overhead_web.json"
```

The experiment results are stored in `reslts` directory of this particular experiment.

E.4 Evaluation Workflow

Each experiment has its dedicated directory under the `/evaluation` directory of the NetShaper repository. To execute the experiment, run `./run.sh` with specific arguments as outlined in the experiment's `README.md` file. The parameters of the experiment controlled by a `json` file stored under `/config` directory of the experiment. Note that these parameters are specific to each experiment, and their descriptions can be found in the experiment's `README.md` file. The results of the experiments, including a `pickle file` and a plot, will be recorded in a subdirectory of the `/results` directory for the experiment, named as the experiment title along with the date and time of the execution. Certain experiments depend on a set of helper scripts for execution, data collection, and result plotting. All these scripts can be found in the `/helper` directory within the experiment.

E.4.1 Major Claims

(C1): NetShaper defeats state-of-the-art classifier with privacy loss (ε) as large as 200. This is proven by the experiment (*E1*) described in section 5.1 whose results are illustrated in fig. 5.

(C2): When shaping is disabled, the peak system throughput and the maximum concurrent clients sustained by the NetShaper middlebox align with those of a direct connection. The NetShaper middleboxes add a small latency overhead ,typically less than 10ms, when compared to a direct connection. This is proven by experiment (*E2*) described in section 5.3. The results of these experiments are illustrated in fig. 7a and fig. 7b respectively.

(C3): When shaping is enabled, the latency of video segments in a video streaming application increases linearly with the DP interval. For all configurations in section 5.4, NetShaper latency is less than $5s$. The bandwidth overhead of NetShaper decreases as the DP intervals become larger. Experiments (*E3*) and (*E4*) reproduce results illustrated in fig. 8a and fig. 8b.

(C4): When shaping is enabled, the latency of webpage requests in a web service increases linearly with the DP interval. The minimum bandwidth overhead occurs at $T = 50ms$. Experiments (*E5*) and (*E6*) prove the results showed in fig. 8c and fig. 8d.

(C5): In both video streaming and web service applications, NetShaper incurs three order of magnitude lower bandwidth overhead than constant-rate traffic shaping. The bandwidth overhead of NetShaper decreases as the number of concurrent flows increases. For video streaming and web service applications, NetShaper requires 11 flows and more than 40 flows, respectively to achieve lower overhead Pacer. Experiment (*E7*) described in section 5.6 proves this, and the results are illustrated in fig. 9a and fig. 9b.

E.4.2 Experiments

In this section, we provide a short description of each experiment. For more details on the experiment steps, preparation, execution, and results, please consult the dedicated `README.md` file of each experiment.

(E1): (Empirical Privacy)[5 human-minutes + 72 compute-hours]: In this experiment, we evaluate the accuracy of state-of-the-art classifiers under the deployment of NetShaper traffic shaping. The runtime depends on GPU architecture, and we used NVIDIA Tesla V100 to train the models. The execution scripts for this experiment are stored under the 'classifier' directory in the NetShaper repository.

(E2): (Middlebox Throughput and Latency)[30 human-minutes + 4 compute-hours]: In this experiment, we measure the maximum throughput that NetShaper middleboxes can sustain, and the latency overhead imposed by NetShaper middleboxes. This experiment is located in the 'microbenchmarks' directory in the project's repository.

(E3): (Video Streaming Latency Overhead)[30 human-minutes + 4 compute-hours]: This experiment evaluates the latency of video streaming application with NetShaper traffic shaping.

(E4): (Video Streaming Bandwidth Overhead)[5 human-minutes + 3 compute-hours]: Measuring the bandwidth overhead of the video streaming application with NetShaper traffic shaping.

(E5): (Web Service Latency Overhead)[30 human-minutes + 2 compute-hours]: In this experiment, we evaluate the latency of web service application.

(E6): (Web Service Bandwidth Overhead)[5 human-minutes + 1 compute-hours]: This experiment measures the bandwidth overhead of web service with NetShaper traffic shaping.

(E7): (Comparison with Related Work)[5 human-minutes + 4 compute-hours]: In this experiment, we compare bandwidth overhead of NetShaper with Pacer and Constant-rate traffic shaping for both video streaming and web service applications.

(E8): (Impact of Privacy Parameters)[5 human-minutes + 1 computer-minutes] In this experiment, we evaluate the effect of the sensitivity (Δ_T), noise (σ_T), and number of DP queues (N) on privacy loss (ε).

E.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.