Beauty and the Burst: Remote Identification of Encrypted Video Streams

Roei Schuster
Tel Aviv University, Cornell-Tech
roeischuster@mail.tau.ac.il

Vitaly Shmatikov
Cornell-Tech
shmat@cs.cornell.edu

Eran Tromer
Tel Aviv University, Columbia University
tromer@cs.tau.ac.il

April 9, 2017

Abstract

The MPEG-DASH streaming video standard contains an information leak: even if the stream is encrypted, the segmentation prescribed by the standard causes content-dependent packet bursts. We show that many video streams are uniquely characterized by their burst pattern and that classifiers based on convolutional neural networks can accurately identify these patterns given very coarse network measurements. We demonstrate that this attack can be performed even by a Web attacker who cannot directly observe the stream, e.g., a JavaScript ad running in the user's Web browser or on a nearby machine.

1 Introduction

Everything has a fingerprint, and so do encrypted video streams. Transport-layer encryption hides the video content but not the network characteristics such as the number of bits transmitted per second. If these features are correlated with content, an adversary who can measure them may be able to identify the video being streamed.

Video streams are known to be bursty [45, 38, 14], and there have been several attempts to use traffic analysis to identify encrypted streamed content [47, 13, 20]. Existing techniques generate a non-trivial number of false positives, make "closed-world" assumptions (i.e., the adversary must know in advance that the streamed video belongs to a small known set), and/or are not robust to noise in the network or the adversary's measurements.

Furthermore, prior work assumes that the adversary can directly observe the encrypted stream either at the network (e.g., a malicious Wi-Fi access point) or physical (e.g., a Wi-Fi sniffer) layer [47, 20], or else that the attacker's virtual machine is co-located with the user's virtual machine [13]. These threat models do not include Web and mobile attackers who can remotely execute some confined code on the user's machine (e.g., a malicious JavaScript ad within the browser).

Our contributions. First, we analyze the root cause of the bursty, on-off patterns exhibited by encrypted video streams. The MPEG-DASH streaming standard (1) creates video segments whose size varies due to variable-rate encoding, and (2) prescribes that clients request content at segment granularity. We demonstrate that packet bursts in encrypted streams correspond to segment requests from the client and that burst sizes are highly correlated with the sizes of the underlying segments.

Second, we demonstrate that this leak is a *fingerprint* for about 20% of YouTube videos because their burst patterns are highly distinct. We also argue that if the streamed video does not belong to the set known to the adversary, it will not be mistaken for one of the known videos. This ensures a

high Bayesian detection rate: if the adversary identifies a streamed video, then this is likely not a false positive.

Third, we develop a new video identification methodology based on convolutional neural networks and evaluate it on video titles streamed by YouTube, Netflix, Amazon, and Vimeo. When trained to recognize 100 titles, our YouTube detector has 0 false positives with 0.988 recall, while the Netflix detector has a false positive rate of 0.0005 with 0.93 recall.

Fourth, we demonstrate that video identification based on burst patterns does not require direct access to the stream. Identification can be done by a remote attacker who serves JavaScript code (e.g., a malicious Web ad) that runs under the confinemenet of the browser's same origin policy. Moreover, the burst patterns can be identified even on a different device: for example, a user watching Netflix on his TV using a Roku streaming device may be attacked by JavaScript code which happens to run in a browser on some PC in the same local network. In both cases, the attack code saturates a network link carrying the targeted video stream and uses the resulting contention to obtain very coarse estimates of the stream's traffic rates—which is sufficient to accurately identify the video. This attacker is much weaker (and the attack thus easier to deploy) than malicious ISPs and Wi-Fi access points typically considered in the traffic analysis literature.

In summary, this paper is the first to (1) explain the root causes of burst patterns in encrypted video streams, (2) show how to exploit these patterns for video identification even in an "open-world" setting, (3) develop and evaluate a noise-tolerant identification methodology based on deep learning, and (4) demonstrate how a remote attacker without direct observations of the network can accurately identify streamed videos.

2 Information Leak in Video Streams

Video streams are bursty. Video streaming traffic is characterized by an initial short period of buffering, followed by the steady state of alternating "On" (short bursts of packets) and "Off" periods—see Figure 2.1. This pattern has been observed for a wide variety of services, devices, clients, and locations [45, 38, 14].

To avoid creating unnecessary traffic, streaming clients typically throttle their content downloads: after the initial buffering, they continue downloading at a speed that is between 1X and 2X the content presentation speed. Clients maintain a target buffer size that is proportional to presentation time and request downloads when the buffer is below the target value.

Streamed video content is typically segmented at the application layer, above the transport and encryption layers. Critically, even if packets are encrypted at the transport layer (e.g., using TLS), their sizes and times of arrival—and, consequently, the sizes of packet bursts and inter-burst intervals—are visible to anyone watching the network. This is a repeated theme in the traffic-analysis literature [47, 19, 21]. If the observable traffic features are correlated with application-layer segmentation, they can leak information about the content of the stream.

MPEG-DASH standard. Modern video streaming services have broadly adopted [10, 8] the MPEG-DASH standard [52, 50] for Dynamic Adaptive Streaming over HTTP (DASH, in short). DASH aims to maximize several measures of quality of experience (QoE) while supporting interoperability with popular streaming technologies. DASH specifies a client-server interface for stream fetching that is independent of the content's bitrate and quality. It does not prescribe any particular fetching discipline, encoding of content, or its presentation. DASH uses TLS for content confidentiality. Content may be additionally encrypted for DRM purposes, but this does not change its network characteristics.

Bursty, on/off behavior of video streams predates DASH, but DASH has effectively standardized it. DASH divides video content into segments based on presentation time and stored in segment-files on the server. Each segment-file corresponds to a particular encoding of a segment. When a streaming session is initiated, the server sends to the client a manifest file referencing the time segments and the available encodings. To obtain the content, the client submits requests for individual segments. The client may request segment-files of any available encoding depending on the presentation considerations and dynamic evaluation of network conditions.

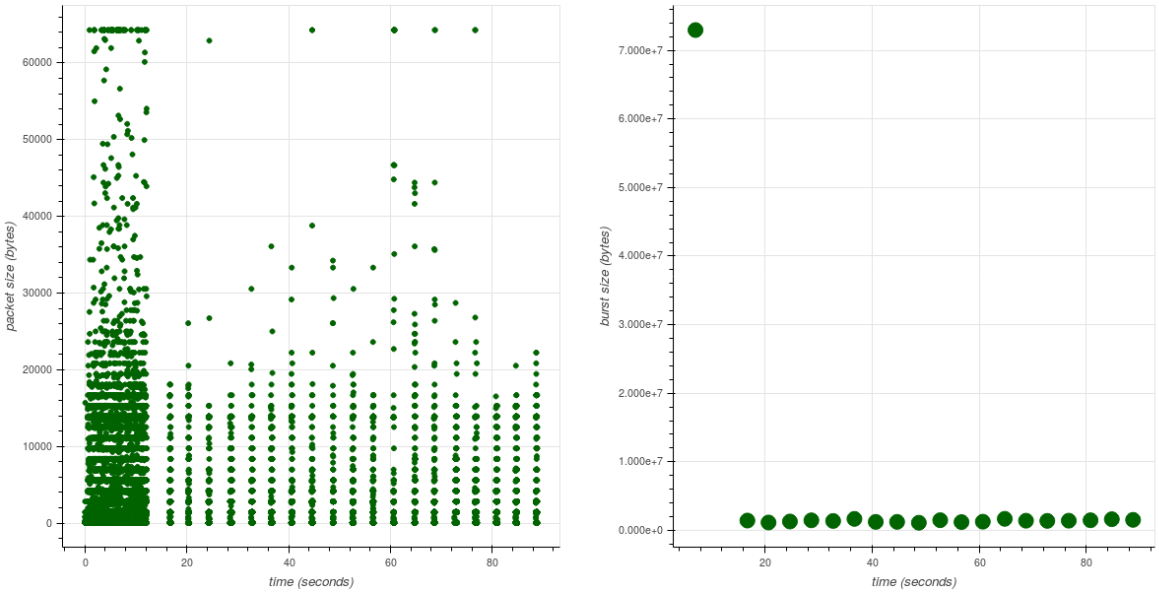


Figure 2.1: Features of a Wireshark capture of Episode 3 of *Mad Men*. The left-hand figure shows packet sizes along the time axis (packet sizes may be larger than Ethernet MTU because of TCP offloading [11])—observe the pattern of buffering followed by the on/off steady state. The right-hand figure shows the size of bursts; the first, largest burst is the buffer.

DASH standardizes a leak. Video compression and encoding algorithms exploit the fact that different video scenes contain different amounts of perceptually meaningful information. All popular streaming services use variable-bitrate (VBR) encoding, where the bitrate of an encoded video varies with its content. Therefore, DASH segments of roughly the same duration (in video-presentation seconds) have very different sizes (in bytes).

DASH video is always streamed in segment-sized chunks. Furthermore, a client requests a new segment when its buffer is just below the target value, and an entire segment finishes downloading long before the client requests another one. Therefore, in a steady-state, on/off stream, burst sizes can help estimate the on-disk segment sizes. The latter sizes, in turn, leak information about the encoded content due to variable-rate encoding. We conjecture that a suffix of the vector of segment sizes, arranged in the order they are fetched from the server (which corresponds to the order of presentation), can be estimated from the observable characteristics of encrypted streaming traffic, up to a small error induced by the varying overheads of lower network layers.

Example. Action scenes, where a lot happens on the screen, are typically encoded with a higher bitrate than non-action scenes. Consider an excerpt from the "Iguana vs. Snakes" scene [5] in the "Planet Earth" series. Figure 2.2 shows how the bitrate of the MP4 file of this video (downloaded from YouTube) changes over time. The video starts with an intense chase scene, featuring an iguana escaping from snakes. In the last 15 seconds, the iguana reaches higher ground and rests on a rock next to another friendly iguana.

To demonstrate this effect more systematically, we created a 45-second "low action" scene by concatenating three copies of the 15-second footage of the resting iguana, and a 45-second "high action" scene by concatenating three copies of another 15-second footage from the height of the chase. We then repeatedly alternated these scenes to craft an artificial 30-minute video, which we uploaded to YouTube (as a private video). We played this video in a Chrome browser configured with an HTTPS proxy to intercept and capture unencrypted content. One of the first HTTPS responses from the YouTube server is a Media Presentation Description (MPD) XML, which describes MPEG-DASH seg-

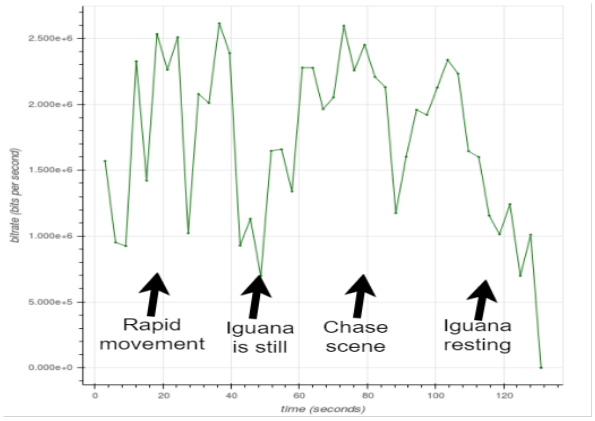


Figure 2.2: Bitrate of the "Iguana vs. Snakes" video.

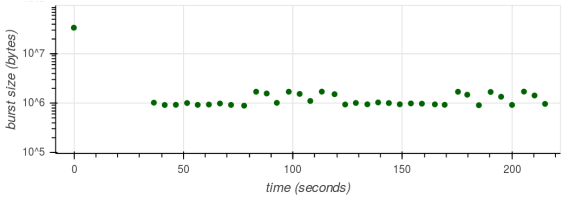


Figure 2.3: Burst sizes when streaming a video with alternating high- and low-bitrate periods. The first, largest burst is the size of the client's buffer.

mentation into 5-second segments. The MPD specifies the audio encoding (135kbps) and five video encoding options corresponding to different resolutions: 144, 240, 360, 480, 720. Subsequent HTTPS responses contain audio and 720p video for the requested segments. Audio and video segment-files corresponding to a given time segment are fetched at roughly the same time, on two respective HTTPS request-response pairs.

As this video is being streamed, we observe the initial buffering period of about 50 seconds, during which segment-files are fetched at a rate higher than their presentation rate. Then the client reaches a steady state and is fetching segment-files exactly every 5 seconds.

We used Wireshark to capture the same traffic encrypted under TLS. Figure 2.3 shows the buffer and burst sizes of the "on" periods in the steady state. During this steady state, when segments are fetched every 5 seconds, burst sizes correspond to the sizes of segment-files. When the segments with an escaping iguana are being fetched, burst size grows. When the segments with a resting iguana are being fetched, it shrinks. Because of the way this video was crafted, "low" and "high" action—and the correspondingly high and low burst sizes—alternate every 45 seconds (9 time segments). In a video stream with different content, the pattern would have been different.

3 Attack Scenarios

3.1 Evaluated attack scenarios

On-path network attacker. If the attacker has passive on-path access to the victim's network traffic at the network (IP) or transport (TCP/UDP) layers, he can directly perform measurements needed for the attack. This includes malicious Wi-Fi access points, proxies, routers, enterprise networks, ISPs, tapped network cables, etc.

Sandboxed JavaScript attacker. Coarse measurements of the victim's stream can also be performed without direct access. The attacker (1) saturates a network link between the victim and the

server, and (2) estimates the fluctuations in the amount of congestion, which indirectly reveals the victim's traffic patterns. This is a special case of timing side channels in schedulers [24, 32] that can be exploited in a variety of attack scenarios.

We focus on remote attackers who can execute JavaScript in the victim's Web browser: rogue websites, advertisers, analytics services, content distribution networks, etc. Their JavaScript is confined by the same origin policy [6], but in Section 9.1 we show how it can carry out the attack if the victim is concurrently streaming video in a different tab or browser instance.

Cross-device attacker. We also consider the case when video is streamed to a different device on the same home network, e.g., a smart TV is playing a movie while a laptop on the same network is running the attacker's JavaScript in a browser. This scenario is similar to the cross-VM scenario, when the attacker's VM is co-located with the victim's VM [13].

3.2 Other attack scenarios

There are many other scenarios where the attacker can indirectly estimate the bitrates and other coarse features of the victim's video stream.

Wi-Fi sniffer. An attacker who is physically close to the victim's Wi-Fi network but not connected to it can set the NIC of his PC or (rooted) smartphone to the promiscuous mode and estimate traffic rates by sniffing physical-layer WLAN packets [59, 15]. If the connection is protected by 802.11, the attacker obtains frames in which all data on top of the media access control (MAC) layer (the lower sublayer of the link layer) is encrypted. This attacker learns the direction of the frames (upstream or downstream) and their sizes. He can also discard MAC-layer management frames as identified by their headers.

Unlike an on-path attacker, a Wi-Fi sniffer cannot distinguish (1) link-layer packet retransmissions and the original transmissions, nor (2) multiple TCP/IP flows on the same link. Both factors introduce some noise into the attacker's observations. Under reasonable network conditions, however, there will be few link-layer retransmissions. We show that our JavaScript attack works even with a noisy, flow-insensitive estimate of the burst size (total number of bytes on the wire)—see Section 9.1. The Wi-Fi sniffing attack should perform at least as well.

Fully remote attacker. A remote attacker who has no foothold in the victim's network can use the same network congestion side channel as our JavaScript attack for coarse-grained traffic measurement [30, 23, 25].

Shared-machine attacker. Our off-path attack is *active*: it requires saturating the victim's link in order to estimate his traffic. If the attacker can execute code on the same machine where the victim is streaming video (e.g., run an app on the same smartphone or even execute JavaScript in the browser), he may be able to estimate traffic via other side channels, such as shared cache or Linux virtual filesystems (sysfs and procfs) [60, 44].

4 Overview of the Attack

Create detectors. For every video file that the attacker wants to identify, he constructs a *detector* algorithm that determines, given measurements of a stream, whether the stream is carrying this video or not.

A detector is created by automatically training a machine-learning model. To generate labeled training data, the attacker streams the video of interest to his own computer and captures the resulting streaming traffic; he also streams other videos as negative examples. This is repeated multiple times (we used up to 100 samples of each video in our experiments) The required capture length depends on the attacker's vantage point: as short as 60 seconds per sample for the on-path attacker, 5 minutes per sample for the JavaScript attacker.[1]

[1] In our experiments, we targeted the first few minutes of the stream, but this generalizes to any sufficiently long section of the video.

Critically, our detectors are **network-agnostic**, because the same segment-files streamed over different networks exhibit the same burst patterns. Therefore, the attacker can train detectors using the data collected on his own network, then use them to identify video streams on the user's network (see Section 7.4).

Since our detectors identify a particular segmented file and not the underlying content, the attacker needs a separate detector for each segmented video he wants to identify. The same content served by different streaming services or different CDN nodes of the same service could have different encodings and segment-files. Moreover, to maximize QoE under varying network conditions, the same content usually has several encodings on the same server (e.g., at different resolutions). YouTube and Netflix support a few dozen encodings [7, 4] but typically no more than 10 per title and device type. The segment-files streamed to the attacker when he is collecting training data must be identical to those streamed to the victim. In practice, we found that Netflix videos streamed on Wi-Fi networks from different ISPs in the same city have exactly identical segmentation (see Section 7.4).

If the attacker's client and network support the highest-quality encoding, he can also induce the service to stream lower-quality encodings either by downgrading through the interface of the streaming application, or by artificially imposing traffic shaping and policy limitations on his network.

Apply detectors. In the online phase of the attack, the attacker measures the victim's network traffic using one of the methods from Section 3. Because video traffic is very distinct and can be accurately recognized from coarse-grained features [59], we assume that the attacker can tell approximately when video playback begins.

He then applies his detectors to the collected measurements to identify the streamed video or determine that it is outside the set of videos for which he has detectors.

5 Experimental Setup

5.1 Network and hosts

As the streaming client, we used Chrome browser running in an Ubuntu 14.04 virtual machine on a Windows host with an Intel i7-3720QM CPU, connected to a university campus network with over 105 Mbps upload and download bandwidth (measured using [9]). We refer to this network as the "training network."

For the cross-network experiments in Section 7.4, we also used a campus Wi-Fi network (10 Mbps upload and download) and a home Wi-Fi network from a cable ISP (82 Mbps). We refer to them as "test networks."

5.2 Data collection

We focused on four popular streaming services: Netflix, YouTube, Amazon, and Vimeo. For our proof-of-concept experiments, we chose a small number of titles from each service. For Netflix, we manually chose 11 popular TV series and up to 10 episodes for each series, totaling 100 titles. For YouTube, we manually chose 20 titles. For each of Amazon and Vimeo, we manually chose 10 titles. See Appendix C for the title list.

Additionally, we crawled YouTube starting from the main page and the front pages of topical channels (e.g., sports and movies) and recursively following recommendation links. The links on the channel front pages are very popular, with over 100k views each. Our crawler thus emulates user behavior: it starts with popular videos and follows YouTube's recommendations. This crawl yielded links to 3,558 videos, to be used in Section 6.

Automated capture. For each title, we spawn a Chrome browser instance to play it, skipping the initial title sequence and starting at the actual content. We employ a service-specific "rewind" procedure to make sure playback begins at the beginning of the content.

The network traffic of each streaming session is captured using Wireshark's `tshark` [12] for the desired duration (see below). For Amazon, Netflix, and Vimeo, the application-layer protocol is TLS; for YouTube, it is either QUIC, or TLS. We will refer to the collected data as *captures* or *captured*

sessions.

Occasionally, playback failed because of Chrome failure or a network glitch. The resulting captures contain very few bytes and we discarded them.

Feature extraction. From each capture, we kept only the TCP flow with the greatest amount of bits on the wire. From each flow, we extracted the time series of the following *TCP flow attributes*: down/up/all bytes per second (BPS), down/up/all packet per second (PPS), and down/up/all average packet length (PLEN). To create uniformly sized vectors, the series were aggregated, by averaging, into chunks of 0.25 seconds.

A *burst* is a sequence of points in a time series (t_i, y_i) such that $t_i - t_{i-1} < I$ for some interval I (0.5 in our experiments). When the points correspond to arrival times and packet sizes, bursts are presumably associated with the transmission of higher-level elements such as HTTP responses (see Section 2). A **burst series** is a series where every point corresponds to a burst. The time of the burst is halfway between the beginning and end of the underlying time-series points that form the burst. The value of the burst is the sum of the values of all of the underlying time-series points. The series were aggregated, by summing, into chunks of 0.25 seconds.

Netflix. We streamed each of the 100 titles one by one and captured the first minute of network traffic for each stream. This was repeated 100 times.

For the cross-network experiments, we chose a subset consisting of 5 episodes of "Mad Men" and 5 other titles. For each title in this subset, we captured 20 90-second streaming sessions on the training network and 20 sessions on the test networks.

YouTube. We streamed each of the 20 titles 100 times and captured 4 minutes of each stream. We had to use longer captures because sometimes the actual content is preceded by an ad, reducing its effective duration.

For each of the 3,558 titles obtained by the automated crawl, we captured a 4-minute streaming session. Unlike the captures of the manually chosen titles, this data is used for measuring the uniqueness of burst patterns, not for the identification experiments. We also downloaded actual 720p MP4 file video files (as opposed to their network streams), using the SAVEFROM.NET Web tool.

Amazon and Vimeo. We streamed every title 100 times. For Amazon, we captured 90 seconds of each stream. For Vimeo, we noticed that burst patterns are very consistent and strongly identifying, so we only needed to capture 60 seconds per stream.

6 From Leaks to Fingerprints

In Section 2, we explained how DASH leaks information about the segment sizes of video files. We now show that for 19% of YouTube files, this leak is actually a *fingerprint*: the sequence of segment sizes identifies the video with virtually no false positives.

Modeling the server. We used the Bento4 MPEG-DASH toolset [1] to process the 3,558 videos from our YouTube dataset (see Section 5.2) for standardized streaming, i.e., divided them into time segments and created the manifests. We opted for 5-second segments, which matches what we observed on both Netflix and YouTube and is close to a recent recommendation [2]. Since the videos were downloaded from YouTube, we expect their encoding parameters to be similar to other YouTube videos. The MPEG-DASH client-server interaction induced by our simulated sever is close to what we empirically observed on YouTube (see Section 2).

Modeling the attacker. Let m be a video. When m is streamed, let its trace $t \in \mathbb{R}^k$ be the sizes (in bytes) of the first k bursts and let T^m denote the probability distribution of these bursts. We assume that T^m is the same whether the video is streamed to the attacker's client (during training) or to the victim's client (during identification). This is empirically justified in Section 7.4.

During training, the attacker acquires n training traces $TS = \{t_1, \ldots t_n\}$ drawn from T^m. Let $s^m = \text{mean}(TS)$. For any $v = (v_1, \ldots v_k) \in \mathbb{R}^k$, define $\alpha(v) \equiv (v_1, \ldots v_k, v_2 - v_1, \ldots v_k - v_{k-1})$. Training produces $\alpha(s^m)$, which is the attacker's *fingerprint* of m. Intuitively, $\alpha(s^m)$ accounts for both the absolute magnitude of segment sizes and their variability pattern.

The attacker is then given $\alpha(t)$ computed from the victim's trace $t \in \mathbb{R}^k$. He concludes that the victim is watching m if and only if $\|\alpha(t) - \alpha(s^m)\|_1 \leq B$, where $B = 3,500,000$ bytes.

Attacker's recall. We wish to compute the recall, or true positive rate (TPR), of the attack. To this end, we first estimate the error $\alpha(t) - \alpha(s^m)$ by lower-bounding the probability that this error is small: $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 < B]$.

We expect that the bigger the burst size, the bigger the potential error. For example, the average size of bursts in the "Iguana vs. Snakes" video is particularly high, over 1MB, vs. the average of 693K across the videos in our set. We streamed this video 100 times, aggregated the traces, and computed the 10-burst fingerprint. We then computed the error for each trace (i.e., the discrepancy between the fingerprint and the trace) and fitted a Gaussian distribution using SciPy's Maximum Likelihood Estimator. The expected value of the error is 41,643 bytes, standard deviation is 24,970 bytes. Observe that $B/7$ is over 10 standard deviations from the expectation of the error. Thus, $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 \leq B/7] \geq 1 - 10^{-12}$, for the aforementioned $k = 10$.

To estimate the error for $k = 40$ (as will be needed later), we partition[2] $t \in \mathbb{R}^{40}$ into 4 contiguous blocks of length 10 and apply the union bound on the probabilities of error in each block, plus the error given by the difference elements in α that are not otherwise accounted for, i.e., $\left|(t_i - t_j) - (s_i^m - s_j^m)\right|$ for $(i,j) \in (11,10),(21,20),(31,30)$. All these sizes are below $B/7$ with probability $\geq 1 - 10^{-12}$. Therefore, we estimate a very high recall probability for $k = 40$ as well, since the error remains tightly bounded below B: $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 \leq B] \geq 1 - 10^{-11}$.

Attacker's precision. Even if the distance between a trace and its fingerprint is small, the attacker may still "misdetect" a video if its fingerprint is close to another one. We show that for almost 20% of the videos in our YouTube dataset, such misdetection is unlikely (and indeed never occurs in practice).

Let D be the 3,558 videos in our YouTube dataset. For $m \in D$, let $z^m \in \mathbb{R}^k$ denote the series of sizes (in bytes) of the first k segments of m, as produced by the server's segmentation of the corresponding MP4 files. We say that a video has *variable segment size* if (1) the overall bitrate is over 100 kBps, and (2) in z^m, more than half of the adjacent pairs differ by more than 110 kB. Let V be the set of videos with variable segment sizes. We observe that in our dataset, $|V| = 671$ ($\approx$19% of D).

A *collision* is a pair of videos $m \in V, m' \in D \cup V$ such that $m \neq m'$, and $\left\|\alpha(z^m) - \alpha(z^{m'})\right\|_1 \leq 2B$. Our dataset does not contain any such collisions given any pair of videos, each represented by 40 segments.

To estimate the attacker's precision, we need to assume that the attacker's fingerprint s^m, which is a series of average burst sizes, is similar to the corresponding series of segment sizes z^m in the following sense: if $\left\|\alpha(z^m) - \alpha(z^{m'})\right\|_1 \geq 2B$, then $\left\|\alpha(s^m) - \alpha(s^{m'})\right\|_1 \geq 2B$. This assumption is empirically true. In general, we expect each burst size to be related to the corresponding segment size by an affine function (accounting for the constant and multiplicative overheads of the encoding and headers added by each network layer).

It follows that no two fingerprints $s^m, s^{m'}$ are $2B$-close in L_1 norm. Since with probability 10^{-11} a trace t (of a video m with variable segment size) is B-close to the correct fingerprint s^m (by the recall bound above), the probability that an attacker mistakes t for another video in our dataset is at most 10^{-11}.

Discussion. This theoretical analysis demonstrates that a significant fraction of YouTube videos are unique given a rudimentary fingerprinting algorithm. This algorithm yields a very strong detector for the videos that satisfy the variable segment size criterion, which is 671 videos out 3,558 in our dataset. Given a video, the attacker can easily check whether it satisfies this criterion.

While our dataset is small in comparison to the entire YouTube, the extremely low error rate and 0 collisions indicate that the attack will generalize. The false positive rate for the the videos satisfying the criterion is very low, which guarantees that the Bayesian detection rate is high even if the base rate is low (see Section 8).

[2]With longer captures, we could have estimated this error directly.

In the following sections, we develop a much more sophisticated and accurate classification method based on machine learning, relax the simplifying assumptions made in the theoretical analysis, and empirically evaluate our method against popular streaming services.

7 Video Identification using Neural Networks

Section 6 explains why DASH-based video streams are fingerprintable, but the theoretical model underestimates the capabilities of realistic attackers who can use traffic features other than burst sizes (e.g., timing). Moreover, the simple classifier based on L_1 nearest-neighbor is clearly suboptimal, e.g., it does not account for the asymmetry of the error distribution. Also, the theoretical model assumes that the attacker can reliably detect bursts and is thus not robust to noisy network conditions.

A more sophisticated classifier would process more and lower-level features and construct a more complex model to characterize the network traces of a given video. In this section, we use machine learning to construct such classifiers. One plausible approach is to compute the classifier of a video from its file, but we found it to be relatively ineffective (see Appendix A). Instead, we use multiple streams of the same content to train a classifier.

7.1 Background on CNNs

Deep learning [35] is a branch of machine learning based on multi-layer artificial neural networks. Deep learning methods have proved very effective for signal recognition tasks such as speech transcription, image segmentation and object recognition, and many others.

In a neural network, each layer of neurons does some computation on its input and passes the output to the next layer (or final output); see Figure 7.1. The first, input layer is a tensor representation of the input, e.g., pixels in the case of image classification. The next (low) levels typically infer representations of the features of the input, and the final (high) layers perform the learning task (e.g., classification) given these features.

Convolutional Neural Networks (*CNNs*) [3] are deep neural networks whose lower layers apply the same linear transformation on many windows in the input data. These layers are typically used to produce representations of local features (e.g., spatially local in an image, or temporally local in a time series). These are suitable for our case, where the network events corresponding to each DASH burst occur in close temporal proximity.

We use supervised training on our neural networks. The training corpus in our case consists of traffic measurements labeled with their correct class, i.e., the identity of the corresponding video. An optimization procedure adjusts the parameters in the functions computed by the layers, so as to minimize the error between the correct classification and the classification produced by the neural network. Learning is only successful if (1) the classifier is able to reduce the training error, and (2) the reduced error rate generalizes to test samples, i.e., inputs that the classifier was not trained on. Training consists of multiple *epochs*. During each epoch, the neural network processes a batch of the training data and its parameters are updated according to the optimization rules.

7.2 Our classifier

We use CNNs with three convolution layers, max pooling, and two dense layers (see Figure 7.1. We train them using an Adam [33] optimizer on batches of 64 samples, with categorical cross-entropy as the error function.

The classifier is constructed using TensorFlow with the Keras front end. For each task, we randomly shuffle the samples, apply the 0.7-0.3 train-test split, and train for a specified number of epochs. The dataset was normalized on a per-feature basis: the time-series vector representing a given feature in each sample was divided by the maximum of the aggregated values of this feature.

Table 7.2 shows the training time, on a workstation with Intel i7-5690X CPU and two NVidia Titan X GPUs. For comparison, we also performed training in an Ubuntu virtual machine on a commodity laptop with an i7-6600U CPU (and no GPUs) running Windows 10; in this case training was 35 times slower, but even so, the most time-consuming training (that of the Netflix classifier for 1,400 epochs) took less than 10 hours.

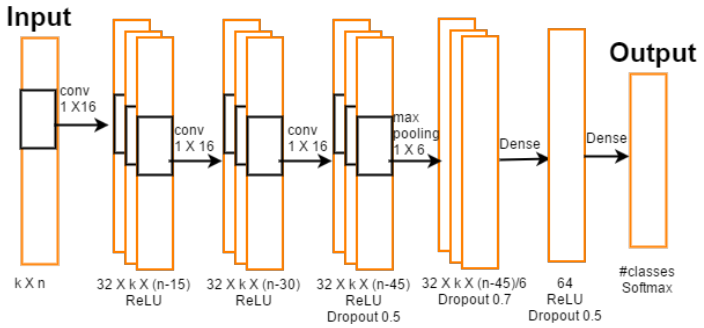


Figure 7.1: Our CNN architecture. k denotes the number of feature types taken. n is the recording time in seconds divided by the time-series sampling rate (0.25).

Dataset	TIME	EPOCHS	PLEN_{IN}	PLEN_{OUT}	PLEN	BPS_{IN}	BPS_{OUT}	BPS	BURSTS	$\text{BURSTS}_{\text{IN}}$	$\text{BURSTS}_{\text{OUT}}$	PPS_{IN}	PPS_{OUT}	PPS
Netflix	497	700	0.318	0.377	0.333	0.983	0.901	0.982	0.926	0.044	0.708	0.917	0.892	0.921
	994	1400	0.301	0.474	0.340	0.983	0.895	**0.985**	0.959	0.949	0.757	0.918	0.881	0.931
YouTube	94	150	0.993	0.993	0.994	**0.995**	0.994	**0.995**	0.984	0.989	0.988	**0.995**	0.993	**0.995**
Amazon	88	700	0.895	**0.925**	0.917	0.899	0.891	0.905	0.790	0.879	0.712	0.792	0.835	0.790
Vimeo	80	500	0.755	0.624	0.741	0.98	0.938	0.984	0.984	**0.986**	0.916	0.958	0.924	0.94

Figure 7.2: Accuracy of our classifiers. TIME is the approximate total training time, in seconds. EPOCHS is the number of epochs. The remaining columns show the accuracy of the classifier when trained on a given feature. The features are the time series of, respectively, packet length, Bps, bursts series, and packets per second (see Section 5.2), measured in the up, down and both directions.

7.3 Classification results

We trained a separate classifier separately for each dataset and each feature type listed in Section 5.2, as well as for each traffic direction (inbound, outbound, or both). Table 7.2 shows the accuracy of these classifiers as the fraction of correctly classified samples.

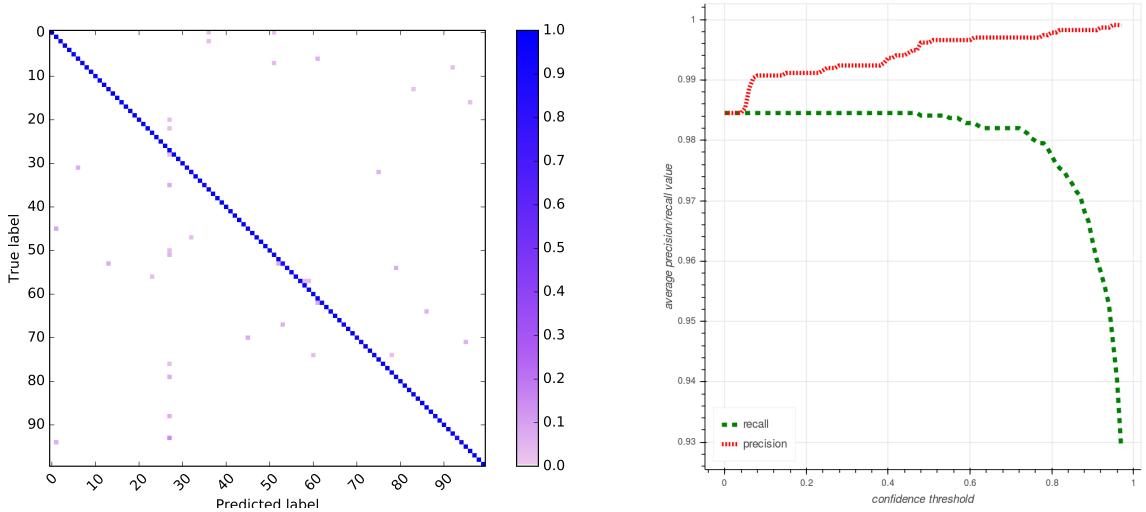
The YouTube classifier is a special case. Not only it achieves 99% accuracy, but it also distinguishes 20 known classes from a large "other" class (unknown videos) with high probability. Furthermore, it works well with any of the features. For example, it achieves 90% accuracy given just the **times** of packet arrivals at a very coarse granularity of 0.25-second intervals (i.e., the PPS feature). This suggests that YouTube streams are particularly susceptible to adversarial identification.

Netflix 1/100 classifier. To gain some insight into how accurate these classifiers are, consider the Netflix classifier that was trained on the BPS feature for 1,400 epochs, achieving 98% accuracy. Figure 7.3a shows the confusion matrix. The classifier does not consistently mistake any class for another. All mistakes but one happen just once. This indicates that different classes do not collide in the classifier's internal representation.

Minimizing false positives. The output of the last, softmax layer of the neural network is traditionally interpreted as a vector of probabilities. The classifier's prediction is the class with the highest probability. We can use this probability as a confidence measure.

Our goal is to ensure that the classifier produces no false positives, at the cost of occasionally failing to detect the match (false negatives). We set a *confidence threshold* and only accept a match if the classifier's confidence is above the threshold. If the confidence is below the threshold, we intentionally misclassify the input as "other" regardless of the class chosen by the classifier.

Figure 7.3b shows the precision and recall of the classifier for various values of the confidence threshold. Precision and recall are calculated by aggregating false positives and false negatives of all classes except "other". Without any loss of recall, we can get a false positive rate of just 0.005 (precision of 0.995). By accepting a 0.07 false negative rate (0.93 recall), we obtain a false positive rate of less



(a) Confusion matrix. Off-diagonal entries are misclassifications. Color in cell i, j denotes the number of samples of class i classified as j.

(b) Precision vs. recall.

Figure 7.3: Netflix 1/100 classifier.

than 0.0005, or a precision of 0.9995, with just 1 false positive out of 2224 matches.

YouTube 1/20 classifier. Our YouTube classifier trained for just 150 epochs on BURSTS achieves 0.994 accuracy. Figure 7.4a shows the confusion matrix. Almost all misclassifications are for "other" (i.e., known titles not recognized), thus there are very few false positives.

Figure 7.4b shows the precision and recall of the YouTube classifier as a function of confirmation threshold. Even when the threshold is 0 (equivalent to simply taking *argmax* of the classifier output), the false negative rate is 0.01 (0.99 recall), and precision is better than accuracy (0.995). By accepting a tiny, 0.002 drop in recall, we achieve **zero false positives**.

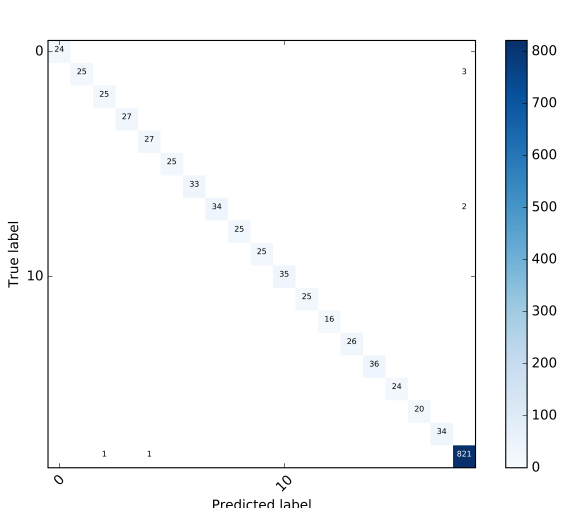
Using multiple feature types. The classifiers discussed above use a single feature type and a one-dimensional input layer ($k = 1$). We also tried more sophisticated classifiers that take in multiple features—in such an architecture, we expect the same one-dimensional layer to pick up localized attributes of different features. We used a greedy search algorithm on the feature set space that begins with an empty set of features and then adds the feature that maximizes validation accuracy after training. Training on multiple features is slower and did not produce significantly more accurate classifiers in our experiments. It is possible that a more elaborate neural network architecture with k independent convolutional layers would work better, albeit with slower training.

7.4 Cross-network training

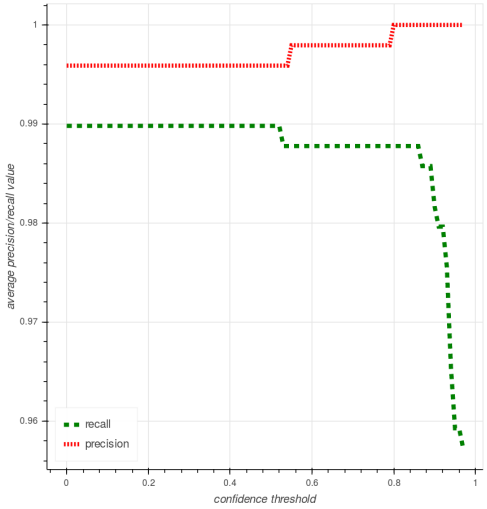
To collect training data, the attacker must stream videos and record traffic. He may be unable to do this on the same local network as the victim, e.g., because that network is secured, or because the attacker wants to identify videos en masse for multiple users on different networks.

The attacker can still perform the training by streaming on his own Internet connection. This connection, however, may have different network characteristics, such as bandwidth, latency, congestions and packet drops, all of which affect the collected traces.

We conjecture that our classifiers learn high-level features of video streams, such as burst patterns, that are robust to reasonable differences in network characteristics and will therefore maintain high accuracy even when trained on a different network (in the absence of pathological conditions such as excessive packet loss or inadequate bandwidth for streaming).



(a) Confusion matrix. Off-diagonal entriess are misclassifications. Number in cell i, j denotes the number of samples of class i classified as j.



(b) Precision vs. recall.

Figure 7.4: YouTube 1/20 + "other" classifier.

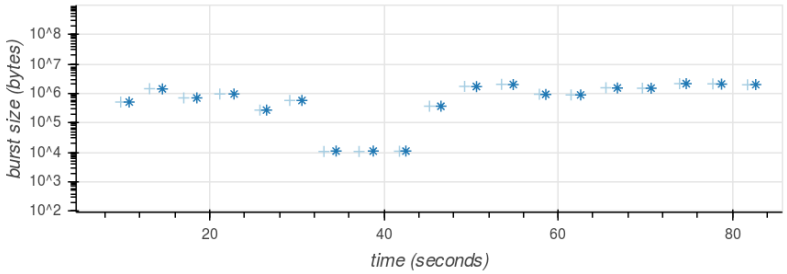


Figure 7.5: Burst sizes of streamed "Reservoir Dogs". The two captures were made on a campus network (+) and a home network (∗).

To confirm this, we captured 90-second streaming sessions of 10 Netflix titles on a campus Wi-Fi network and on a home Wi-Fi network from a cable ISP. We trained our classifier on the campus data and measured its accuracy on the home-network data. Our classifier uses only the down BURST series (see Section 5.2). Trained on 50 campus captures per title, it reaches 98% accuracy on the home-network data (20 captures per title). Figure 7.5 shows that the burst patterns on the two networks are highly correlated and aligned in time.

8 Bayesian Detection Rate

In Section 6 and Section 7, we showed classifiers with very low false positive rates. However, the attacker's false alarm rate is not the detector's false positive rate but the *Bayesian Detection Rate (BDR)*. The BDR of a detector for video m is the probability $\Pr(M|A)$, conditioned on the detector declaring that m is being streamed (event A), that the victim is indeed streaming m (event M). This probability is taken over all videos that the victim could be streaming, as well as network conditions and measurement noise.

$\Pr(M|A) = \frac{\Pr(A|M)\Pr(M)}{\Pr(A|M)\Pr(M)+\Pr(A|\neg M)\Pr(\neg M)}$ by Bayes's Law. We can estimate $\Pr(A|M)$ by the detector's recall, and $\Pr(A|\neg M)$ by its false positive rate.

"Open world," when the attacker does not know a priori a relatively small set of possibilities for the video that he aims to identify, is characterized by an extremely low *base rate*, i.e., low probability $P(M)$ that the victim is actually streaming the video corresponding to any of the classifiers the attacker is applying. In this setting, when the attacker's recall is sufficiently high, the BDR is dominated by the false positive rate.[3]

We now revisit the detectors from the previous sections and explain how they are expected to perform in the "open-world" setting.

8.1 Distance detector

We first analyze the BDR of the detector from Section 6.

Let $\hat{D}$ be the world of videos, and let $\hat{V} \subseteq \hat{D}$ be the world of videos with variable segment size. $\psi^{\hat{D}}$ is the distribution over $\hat{D}$. Assume that the victim chooses $m' \leftarrow \psi^{\hat{D}}$, and that the videos on our server D were likewise sampled from $\psi^{\hat{D}}$ (i.e., by sampling videos according to their popularity on the service). As before, let V be the subset of D consisting of videos with variable segment size. The attacker monitors some $m \in V$. Let $t \leftarrow T^{m'}$ be the trace received by the attacker.

Let COL denote the collision event $\left\|\alpha(s^m) - \alpha(s^{m'})\right\|_1 \leq B$. If the detector misclassifies t (i.e, declares a match but $m \neq m'$), then either COL, or $\left\|\alpha(s^{m'}) - \alpha(t)\right\|_1 \geq B/2$. The latter has very low probability since the recall is very high, $> 1 - 10^{-11}$). We now bound p_{COL}, the probability of COL (i.e., false positive rate).

If this probability is high, $p_{\mathsf{COL}} \geq \frac{2}{10^6}$, then we are likely to observe a collision in our dataset. Under the simplifying assumption that collisions in our dataset are independent events,[4] then with overwhelming probability of $1 - (1 - p_{\mathsf{COL}})^{(|D|-|V|)|V|+|V|^2/2} > 0.9862$) there exist $m_V \in V, m_D \in D$ such that $m_V \neq m_D$ and $\|\alpha(s^{m_D}) - \alpha(s^{m_V})\|_1 \leq B$. Since we did not observe any such collisions in 2,162,297 pairwise tests over our dataset, it is likely that $p_{\mathsf{COL}} \leq \frac{2}{10^6}$.

In this case, assuming the open-world base rate is $\frac{2}{10^6}$, the BDR is very close to 0.5.

8.2 Neural-network detector

YouTube. With our YouTube classifier, when we preferred precision over recall, there were no false positives: we never observed an "other" video that was misclassified as one of the known videos. We view this as an indication that our results generalize.

Netflix. With our Netflix classifier, when we preferred precision over recall, we observed 1 false positive (compared to 2,240 true positives), corresponding to a false positive rate of 0.00045. Our recall is still > 0.93.

At first glance, this result seems harder to generalize. We cannot simply plug in $\Pr(A|\neg M)$ and $\Pr(A|M)$ to the BDR formula and expect to get a good estimation, since the distribution that this classifier was trained on textemdash without samples from the catchall "other" class—is fundamentally different from the distribution of videos that might be streamed by the user.

Similarly to the previous section, there are two causes of false positives: similarities in the videos' burst patterns (which is what the classifier learns), i.e., a *classifier collision false positive*, and noise in the measurements, i.e., a *network noise false positive*.

The confusion matrix (Figure 7.3b) shows no pairwise classifier collisions for the 100 titles. The classifier does not consistently confuse any particular title for another (even though many are episodes

[3]For example, even if the recall is $\Pr(A|M) = 1$ and the false positive rate is $\Pr(A|\neg M) = \frac{1}{1000}$, in a setting where the victim iteratively chooses one out of 100,000 videos and the attacker has a detector for only one of them, we would expect the attacker to get roughly 100 false matches before he sees one true match.

[4]This assumption is an approximation. It could have been strongly violated, e.g., if all collisions were due to a small set Z of videos that each collides with many other videos: as long as we didn't hit any of Z when picking D, we would not observe any collisions. However, due to the geometrical structure of video fingerprints, this seems unlikely. The existence of videos in Z whose fingerprints are close to those of many other videos, implies that those videos also have fingerprints that are geometrically close to each other and are thus likely to collide in D.

in the same TV series with presumably similar visual attributes). We conclude that classifier collisions are uncommon.

When collisions do not occur, our classifier, tuned for precision rather than recall, performs very well: for 2,224 validation samples, it only misclassifies one. No other sample was close enough, in the classifier's eyes, to *any* of the 99 classes. This means that the classifier made one "confident" mistake out of, possibly, 220,176.

Moreover, the Netflix classifier was trained on just 60-second captures. This is equivalent to only about 45 seconds of steady-state bursts, after accounting for the (less discriminative) buffering period. We expect that it may be possible to train an even more powerful classifier by increasing the length of the captures to 90 seconds.

9 Remote Attacks

9.1 Measurement with JavaScript

We now consider a remote attacker who only has a restricted foothold in the user's network. For example, he controls an ad embedded in some webpage visited by the user. Such ads are typically sandboxed because they may come from questionable sources with strong commercial interest in users' data (including their viewing habits).

An ad may include JavaScript code executing in the user's browser, but this code is confined—both by the main browser sandbox, which prevents it from issuing arbitrary requests to the OS, and by the same origin policy [6], which prevents it from reading the content that belongs to other Web origins. In particular, even if the user is streaming some video in another tab of the same browser, confined JavaScript code cannot directly access the URL or content displayed in that tab.

The same origin policy does permit JavaScript code to communicate with its own origin (e.g., the Web server that served the ad). This communication is carried over the same Internet connection as the streaming video. Since Internet links usually have bounded bandwidth capacity, this means that the malicious JavaScript code and Web content from other origins (including video) share a limited resource. JavaScript code can send and receive arbitrary amounts of data from its colluding server to create *artificial congestion* on the shared link.

When the shared link is congested, any attempt to use it can create observable delays in the malicious code's own communication. The malicious code can then estimate how much traffic is flowing over the link by measuring delays in its communication with its server. This leaks information about the content streamed from a different origin by the same browser (a **cross-site attack**, see Figure 9.1), or even by a different device on the same local network (a **cross-device attack**, see Figure 9.3).

9.2 Cross-site Attack

Simulating the attack. As a proof of concept for the cross-site video identification, we implemented a malicious Node.js Web server which, when accessed by the victim's browser, serves *detector code* written in JavaScript. This code, running unprivileged within the browser sandbox, talks back to the server via the Socket.IO API. The server sends a stream of messages, causing congestion. The detector code measures the arrival time of these messages, using `window.performance.now()`, to detect contention from other traffic on the shared link.

We simulated this attack in a Chrome browser running in a Ubuntu 14.04 VM, with a simulated 45 Mbps (5.625 MBps) down/upstream bandwidth (capped by VMWare Workstation). The server was on the same LAN as the victim machine. The attack server's messages contain 6 KB of random data, sent at the rate of 1 per 0.001 seconds and an overall transmission rate of 6 MBps. This is more than enough to saturate the simulated link. [5]

[5] A portion of messages is queued at the server. The server's CPU and memory utilization remain very manageable: an Intel Core i7-5960X CPU is utilized at around 30% of one core, and less than 500MB of memory over the attack's duration. More careful calibration of the transmission rate, using flow control, would have reduced server load while maintaining the same attack effectiveness.

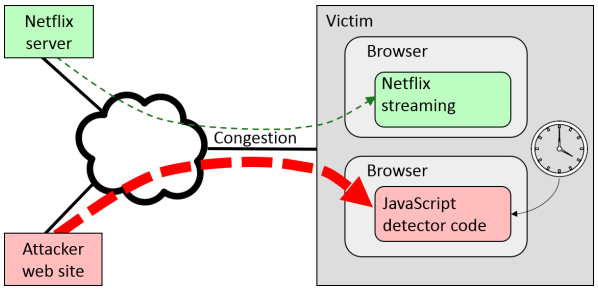
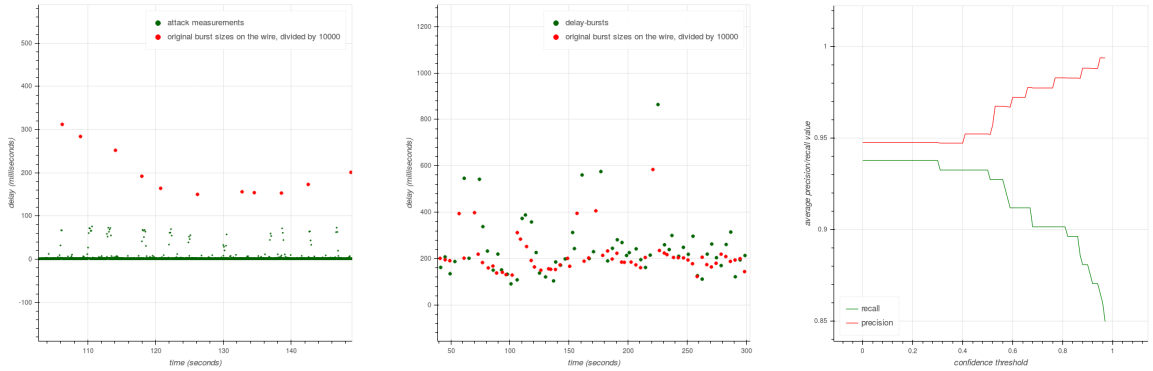


Figure 9.1: Adversary model of cross-site attack.



(a) Actual bursts size vs. measurements from JavaScript in a different origin.

(b) Delay bursts vs. actual bursts (burst are in units of 10^4 bytes).

(c) Precision vs. recall of classifier

Figure 9.2: Cross-site JavaScript attack performance.

We opened two browser windows, one streaming the selected video, the other executing the JavaScript client, and repeated the experiment 100 times. Figure 9.2a shows that bursts in the video stream are very visible in the measurements performed by the JavaScript client.

Data. We evaluated the JavaScript attack on 10 Netflix titles: 5 episodes of the first season of "Mad Men" and 5 arbitrarily chosen other titles. For each of these titles, we collected 100 captures of 5 minutes of playback, as measured from concurrently executing JavaScript.

Let $\{X_n\}$ be the vector of message arrival times for an attack session. We compute $Y = (0) \| ((X_2, \ldots X_n) - (X_1, \ldots X_n - 1))$, the vector of message delays, and filter the X, Y time series for points where the delay is $y > 8$ms. From this, we compute the burst series (see Section5.2) with 0.25 second intervals. We then filter out all bursts whose value is below 80. We call the resulting time series the *delay bursts* of the attack session. To fit the delay burst series into the classifier, we chunked it into 0.25 intervals.

Classifier. We use a similar classifier to the one in Section 7.2, with a few minor modifications. Using noisier and longer training captures cause the classifier to overfit, so we applied more aggressive regularization by adding two more dropout layers after the first and second hidden layers. We also changed the convolution dimension from 16 to 12 and the max pooling dimension from 6 to 3. Finally, we used Adadelta instead of the Adam optimizer.

Results. As Fig. 9.2b shows, the delay bursts series is strongly correlated with the bursts of the concurrent stream. Our 1/10 Netflix classifier attains 94% accuracy. As in Section 7, we can tweak our confidence threshold to reduce false positives at the cost of reducing recall (see Fig. 9.2c). For example, by accepting 0.85 recall, we obtain precision of 0.994.

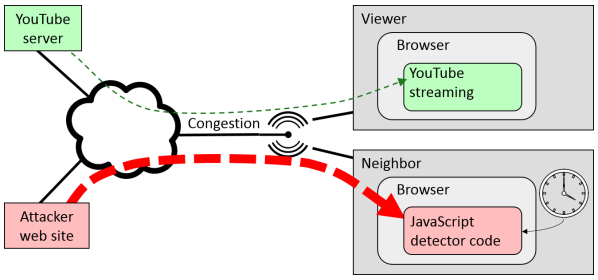


Figure 9.3: Adversary model of cross-device attack.

9.3 Cross-device attack

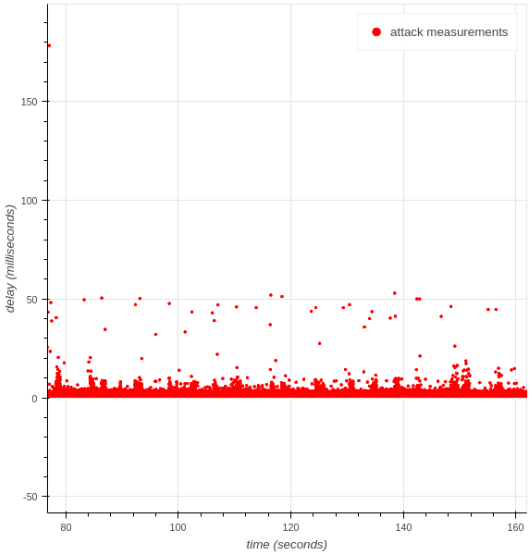
We now investigate feasibility of cross-device attacks, where sandboxed JavaScript code running on a *neighbor* device identifies videos streamed to a *viewer* device on the same local network (see Figure 9.3). We employ a setup common in home WiFi networks. We find that results are similar to the case of cross-site attack: delay-burst sizes (computed from traffic measurements) correspond to traffic burst sizes, and a neural network can distinguish among titles based on these measurements alone.

Experimental setup. As the viewer device, we used a laptop (Intel i7-5600U CPU) running Ubuntu 16.04. As the neighbor device, we used a laptop (Intel i7-3720QM CPU) running an Ubuntu 14.04 VM guest in a Windows host. Both were connected over Wi-Fi to an Asus RT-AC66U wireless router, connected to a university campus Internet. Similarly to Section 9.2, to systematically simulate Internet connectivity with limited bandwidth, the router was configured to cap its total downlink speed at 45 Mbit, using the "Max Bandwidth Limit" setting of the Tomato Advanced firmware. The attack server was configured as in Section 9.2, but sending 8kB per 1.5 milliseconds. This is about 300 kBps short of saturating the network link.

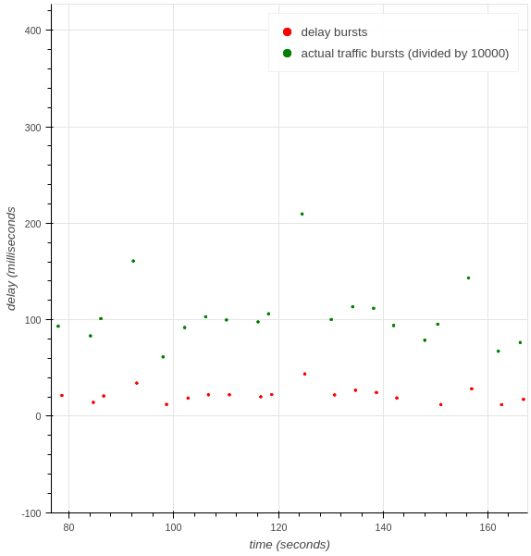
Data. We alternately streamed the first and second episodes of Mad Men to the viewer, 100 times for each episode, and recorded the traffic delays observed by the neighbor. We smoothed the time-series of the delay measurements by averaging over 0.1 second intervals. We then computed the delay bursts of the smoothed attack measurements, similar to Section 9 except for the parameters: the time series was filtered for delays $y > 1.9$ms, the burst series was computed with 0.5 second intervals, and we filtered out all bursts whose value was below 10. To fit the delay burst series into the classifier, we chunked it into 0.25 intervals.

Results. The timing of the messages observed by the detector code on the neighbor device exhibits clear patterns corresponding to the streaming traffic sent to the viewer device. Figure 9.4a shows that a streaming segment during the steady state caused delays in the messages received by the neighbor. Figure 9.4b shows delay bursts, with a clear visible correlation to segment sizes (traffic bursts). Unsurprisingly, our classifier from Section 9.2 performs well on these data as well, attaining 95% accuracy differentiating between the two episodes.

Measurements on TV streaming player. Nowadays streaming video content is often watched on a smart TV, or via a streaming device that connects to a TV screen via a HDMI connection. To investigated the traffic characteristics during such streaming, we used a setup similar to above except that the viewer was a Roku Premiere streaming device (a very popular brand), connected to the Internet via WiFi. The video segment bursts are, gain, clearly observable. For example, Figure 9.5 show the attacker's measurements and corresponding delay bursts, while streaming episode 1 of Mad Men. The expected pattern, a large burst followed by smaller ones in steady intervals, is clearly discernible. We interpret these preliminary results as a promising indication that video identification can be feasibly performed by a neighbor attacker on a Roku user.

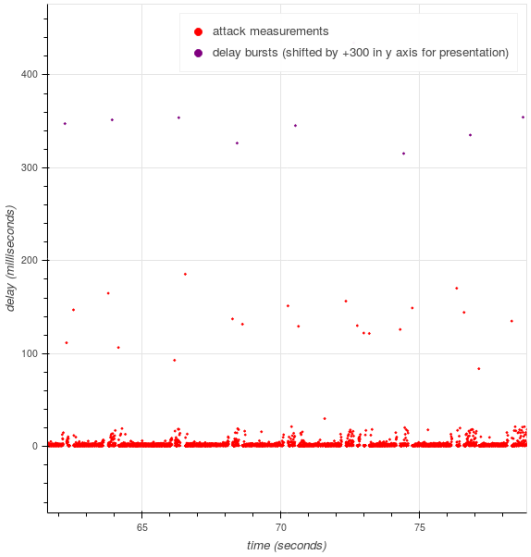


(a) Raw attack measurements, showing delays at roughly steady intervals.

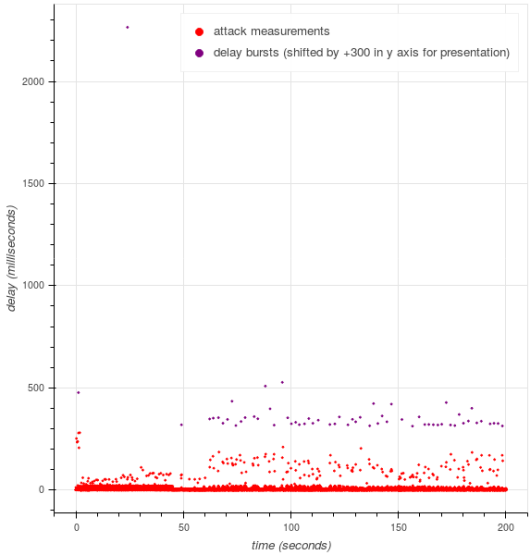


(b) Delay bursts vs. actual traffic bursts. Actual bursts were divided by 10000 for presentation.

Figure 9.4: Message delays observed by the neighbor in a cross-device attack.



(a) Local view during steady-state streaming. Streaming bursts, at steady intervals, cause a momentary increase in delays, or "delay bursts".



(b) Global view, including initial buffering.

Figure 9.5: Message delays observed by the neighbor in a cross-device attack against a Roku device.

10 Limitations

Our attack relies on two assumptions: (1) the attacker can measure traffic bursts in the victim's video stream, and (2) the pattern of these bursts is similar to what the attacker observed when streaming the same title.

The attack works well using only very coarse traffic features (see Section 7.3) and is therefore robust to minor noise in the stream or in the attacker's measurements. If the noise is so significant as to dramatically change the traffic characteristics of the stream (e.g., if the same network connection is used to watch multiple concurrent videos, upload media files, or for some other bandwidth-intensive activity), the attack may not succeed.

In the JavaScript attack, the server sends large amounts of traffic to congest a shared network link and the client measures arrival times in the victim's browser. If the client code does not have access to precise time, the roles must be reversed (see Section 12). The ability of malicious JavaScript in the victim's browser to congest the network may be limited by resource-intensive processes executing on the same machine.

As explained in Section 4, different encodings of the same content create different burst patterns. The attack will not succeed if the encoding of the streams used to train the attacker's detector is different from the encoding of the victim's stream.

Our techniques aim to identify standard, unmodified streaming video (e.g., Netflix movies). They are not designed to resist evasion. If the user or service re-encodes the video (e.g., at a different resolution), the attacker's previously trained detectors will no longer work.

11 Related Work

11.1 Exploiting VBR leaks

Fine-grained video. Saponas et al. [47] observed that encrypted, VBR-encoded videos leak information about their content. To create a "signature" of a video, they take its traffic trace as a bits-per-second time series at the granularity of 100 milliseconds, average, and apply a sliding-window DFT. Their detector applies DFT to traffic traces and matches to the closest signature.

Li et al. [36] focus on redistributed, DRM-protected content that has been re-encoded to avoid detection. They apply a wavelet transform on the time series of frame sizes. Their detector cross-correlates the wavelet coefficient series of observed traffic with those of a reference content file. In [37] Liu et al. improve this method by using aggregated traffic throughput traces (as opposed to the frame-size time series). They report 1% false positive rate and 90% recall rate.

These methods are not applicable to modern streaming services. They operate on time series resembling, and close to the granularity of, the sizes of individual video frames. DFTs and wavelet transforms capture short-term variations caused by changes of picture and long-term variations caused by changes of scene. In our setting, the observable features are not individual frames but bursts 4–6 seconds apart, or typically 120–180 frames.

Even though these methods rely on fine-grained measurements, their false positive rates are prohibitively high in an open-world setting (due to the base rate fallacy, even 1% false positive rate implies an extremely low Bayesian Detection Rate). None of them would work if the measurements of the attacker (e.g., sandboxed JavaScript code) are noisy and coarse-grained.

Dubin et al. [20] suggest using the (unordered) set of segment sizes as a title fingerprint. This assumes that segment sizes uniquely characterize a video and can be measured precisely. This detector is far less accurate than our classifiers and vulnerable to noise, and consequently cannot be used by a JavaScript attacker. See Appendix B for the detailed analysis.

VoIP. Wright et al. showed that VBR leakage in encrypted VoIP communication can be used to identify the speaker's language [57] and detect phrases [56]. Their detector is a Hidden Markov Model trained to identify a specific phrase. White et al. [55] extended this approach to extract conversation transcripts.

11.2 Congestion and timing attacks

The general approach of creating congestion on a shared resource (network, in our case) and using it to measure a concurrent process's consumption of that resource is used, for example, in shared-cache attacks on cryptographic computations [46, 41, 28].

Our network congestion attack works because traffic-flow scheduling policies for a shared internet link are leaky. Kadloor et al. [24, 31, 32] studied the tradeoffs between delays, fairness, and privacy in scheduling policies on shared resources. Kadloor et al. [30] also showed how to exploit the queueing policy in DSL routers: by sending a series of ICMP echo requests (pings) and timing RTTs, they infer the traffic patterns of a remote user. This attack can also help infer the website being visited [23, 25]. This attack is very powerful because the attacker only needs to know the user's IP address, but it cannot be deployed if the user is behind a firewall or a router that discards unsolicited packets from outside the network (as many routers do by default, nowadays).

Agarwal et al. [13] show how a VM can use link congestion to infer the traffic patterns of a co-located VM.

To the best of our knowledge, the ability of confined JavaScript to perform network measurements at sufficient granularity to identify video streams has never been empirically demonstrated before. This is a particularly dangerous scenario because untrusted JavaScript code from operators who have commercial interest in users' viewing habits is ubiquitous on the Web.

Timing attacks have a long history in computer security [51, 17]. Felten and Schneider [22] observed that JavaScript can infer information from the timing of cross-origin requests; Bortz and Boneh [16] demonstrated several timing-related Web attacks; Van Goethem et al. [53] propose timing techniques that tolerate network noise and proposed server-side mitigations. Oren et al. [40] used JavaScript timing mechanisms for a cache attack. Kohlbrenner and Shacham [34] showed that existing browser-based mitigations are insufficient and proposed a new browser-based defense.

11.3 Fingerprinting and traffic analysis

There is a large body of research on identifying websites in encrypted network traffic [26, 19, 27, 23, 18, 49, 54, 43]. Juarez et al. [29] argue that most of these efforts make unrealistic assumptions and fail to cope with the base rate fallacy. Panchenko et al. [42] evaluate a state-of-the-art method for web*site* detection and conclude that web*page* detection is infeasible. Traffic analysis was used to infer application-specific sensitive information, such as health conditions [39, 19], as well as Web sources of video traffic [48]. Prior work also includes mitigations [58] and counter-mitigations [21].

12 Mitigations

Segment size leak. The root cause of information leaks in video streams is that, for any sufficiently long video, the encoding bitrate changes over the presentation time in a unique, identifying way. Segmenting video files and transmitting them in bursts (primarily done to maximize quality of experience) reduces the granularity of the leak but does not prevent videos from being identified.

Decreasing granularity further, to minutes, will not entirely prevent the leak in longer videos, but will degrade QoE and network efficiency. Segmenting VBR video into uniformly sized segments is futile because then their *duration* will differ, thus the timing of client requests will still leak similar information.

Constant-rate encoding with tight rate control and large segments will eliminate the leak, but at the cost of a very inefficient encoding. Similarly, padding bursts to the maximum segment size would require transmitting much more traffic than the actual file size.

The VBR pattern is inherently observable in traffic if the duration of the client's buffered video is close to constant (or, more generally, an affine function of presentation time). Solving the problem thus requires a different buffering regime, but devising such a regime is non-trivial. For example, consider a *variable-size buffer* that fetches equally-sized segments every X seconds (where X is fixed). This necessarily requires a balance between some degree of rate control (lest the buffer runs out in the middle of long, complex action scenes) and an increase in the initial buffering time (to maintain

robustness to network conditions while also accounting for the possibility of sudden buffer depletion due to high-bitrate content). Both factors would directly degrade user experience and network efficiency.

Network congestion side channel. The malicious traffic required for the congestion attack may be recognizable at the network level. Detection and prevention mechanisms can thus be placed at the router, network, OS, or browser. Fuzzy-time sandbox solutions such as [34] would not entirely prevent our attack: the JavaScript client can still send packets to congest the uplink, yet timing measurements can be performed by the colluding server.

13 Conclusions

Leakage of information about video content via network traffic patterns is prevalent in modern streaming protocols and popular services. We implemented and evaluated a novel method based on deep learning that exploits this leak for video identification.

Our method is tuned for high precision and effective in an "open-world" setting. It can be used by on-path adversaries such as ISPs and enterprise networks to spy on their users. Furthermore, it exposes sensitive information of the streaming service itself. For example, ISPs can use it to construct a popularity histogram of Netflix videos (Netflix does not release this information). We also show how an adversary who merely serves a Web page or ad to a user can, via the network congestion side channel, perform the measurements needed for the attack and identify the videos being streamed by the user.

Acknowledgements. This work was supported by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by the Check Point Institute for Information Security; by Google Faculty Research Awards; by the Israeli Ministry of Science and Technology; by the Israeli Centers of Research Excellence I-CORE program (center 4/11); by the Leona M. & Harry B. Helmsley Charitable Trust; and by National Science Foundation grant 1612872.

References

[1] Bento4 MPEG-DASH tool set. `https://www.bento4.com/developers/dash/`. Accessed: 2017-01-16.

[2] Bitmovin. `https://bitmovin.com/mpeg-dash-hls-segment-length`. Accessed: 2017-01-16.

[3] Convolutional neural networks. `https://en.wikipedia.org/wiki/Convolutional_neural_network`. Accessed: 2017-01-16.

[4] Netflix tech blog: Per-title encode optimization. `http://techblog.netflix.com/2015/12/per-title-encode-optimization.html`. Accessed: 2017-01-16.

[5] Planet earth II: Iguana vs snakes. `https://www.youtube.com/watch?v=Rv9hn4IGofM`. Accessed: 2017-01-16.

[6] Same origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`. Accessed: 2017-01-16.

[7] Stackoverflow: Youtube encoding. `http://video.stackexchange.com/questions/5318/how-does-youtube-encode-my-uploads-and-what-codec-should-i-use-to-upload`. Accessed: 2017-01-16.

[8] The state of MPEG-DASH deployment. `http://www.streamingmediaglobal.com/Articles/Editorial/Featured-Articles/The-State-of-MPEG-DASH-Deployment-96144.aspx`. Accessed: 2017-01-16.

[9] testmy: web-based bandwidth test. `http://testmy.net/`. Accessed: 2017-01-16.

[10] Why YouTube and Netflix use MPEG-DASH in HTML5. https://bitmovin.com/status-mpeg-dash-today-youtube-netflix-use-html5-beyond/. Accessed: 2017-01-16.

[11] Wikipedia: TCP offload engine. https://en.wikipedia.org/wiki/TCP_offload_engine. Accessed: 2017-01-16.

[12] Wireshark. https://www.wireshark.org/. Accessed: 2017-01-16.

[13] Yatharth Agarwal, Vishnu Murale, Jason Hennessey, Kyle Hogan, and Mayank Varia. Moving in next door: Network flooding as a side channel in cloud environments. In *International Conference on Cryptology and Network Security*, pages 755–760. Springer, 2016.

[14] Pablo Ameigeiras, Juan J Ramos-Munoz, Jorge Navarro-Ortiz, and Juan M Lopez-Soler. Analysis and modelling of YouTube traffic. *Transactions on Emerging Telecommunications Technologies*, 23(4):360–377, 2012.

[15] John S Atkinson, O Adetoye, Miguel Rio, John E Mitchell, and George Matich. Your wifi is leaking: Inferring user behaviour, encryption irrelevant. In *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*, pages 1097–1102. IEEE, 2013.

[16] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

[17] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[18] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 605–616. ACM, 2012.

[19] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 191–206. IEEE, 2010.

[20] Ran Dubin, Amit Dvir, Ofer Hadar, and Ofir Pele. I know what you saw last minute — the chrome browser case. In *Black Hat Europe 2016*, 2016.

[21] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 332–346. IEEE, 2012.

[22] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32. ACM, 2000.

[23] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 58–78. Springer, 2012.

[24] Xun Gong and Negar Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *Biological Cybernetics*, 24(3):1841–1852, 2016.

[25] Xun Gong, Negar Kiyavash, and Nikita Borisov. Fingerprinting websites using remote traffic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 684–686. ACM, 2010.

[26] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42. ACM, 2009.

[27] Andrew Hintz. Fingerprinting websites using traffic analysis. In *International Workshop on Privacy Enhancing Technologies*, pages 171–178. Springer, 2002.

[28] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[29] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 263–274. ACM, 2014.

[30] Sachin Kadloor, Xun Gong, Negar Kiyavash, Tolga Tezcan, and Nikita Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5. IEEE, 2010.

[31] Sachin Kadloor and Negar Kiyavash. Delay optimal policies offer very little privacy. In *INFOCOM, 2013 Proceedings IEEE*, pages 2454–2462. IEEE, 2013.

[32] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *Biological Cybernetics*, 24(3):1562–1573, 2016.

[33] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480. USENIX Association.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[36] Yali Liu, Canhui Ou, Zhi Li, Cherita Corbett, Biswanath Mukherjee, and Dipak Ghosal. Wavelet-based traffic analysis for identifying video streams over broadband networks. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–6. IEEE, 2008.

[37] Yali Liu, Ahmad-Reza Sadeghi, Dipak Ghosal, and Biswanath Mukherjee. Video streaming forensic–content identification with traffic snooping. In *International Conference on Information Security*, pages 129–135. Springer, 2010.

[38] Jim Martin, Yunhui Fu, Nicholas Wourms, and Terry Shaw. Characterizing netflix bandwidth consumption. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 230–235. IEEE, 2013.

[39] Brad Miller, Ling Huang, Anthony D Joseph, and J Doug Tygar. I know why you went to the clinic: Risks and realization of https traffic analysis. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 143–163. Springer, 2014.

[40] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.

[42] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. Website fingerprinting at internet scale. In *Network & Distributed System Security Symposium (NDSS). IEEE Computer Society*, 2016.

[43] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.

[44] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM, 2012.

[45] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network characteristics of video streaming traffic. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 25. ACM, 2011.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[47] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, Tadayoshi Kohno, et al. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *Usenix Security*, volume 3, page 3, 2007.

[48] Yan Shi and Subir Biswas. Protocol-independent identification of encrypted video traffic sources using traffic analysis. In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[49] Yi Shi and Kanta Matsuura. Fingerprinting attack on the tor anonymity system. In *International Conference on Information and Communications Security*, pages 425–438. Springer, 2009.

[50] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[51] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, volume 2001, 2001.

[52] Thomas Stockhammer. Dynamic adaptive streaming over HTTP: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[53] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[54] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 201–212. ACM, 2013.

[55] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 3–18. IEEE, 2011.

[56] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 35–49. IEEE, 2008.

[57] Charles V Wright, Lucas Ballard, Fabian Monrose, and Gerald M Masson. Language identification of encrypted voip traffic: Alejandra y roberto or alice and bob? In *USENIX Security*, volume 3, page 3, 2007.

[58] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9, 2009.

[59] Fan Zhang, Wenbo He, Xue Liu, and Patrick G Bridges. Inferring users' online activities through traffic analysis. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 59–70. ACM, 2011.

[60] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

A Streams vs. MP4 files

Since the cause of the leak is the DASH standard, it is plausible that a detector can be computed directly from a video file.[6] This would dramatically reduce the attacker's workload because he would not longer have to stream each video multiple times.

There are challenges, however. First, it is necessary to infer the exact parameters used for segmentation, such as segment duration and minimal buffer time, and how they change with respect to file encoding, size, bitrate, view count, etc. Each service may have many combinations of these parameters because they change over time and, possibly, do not affect already-segmented files. Second, this approach would not work at all if the attacker does not have the file (as in the case of Netflix).

To learn the relationship between MP4 files and streams, we would like to train a classifier that takes in an MP4 file and a traffic capture, and outputs whether the latter is a stream of the former. We used our dataset of 3,558 YouTube videos for which we have both the files and the captures. The first goal is to align the stream with the file, i.e., align traffic bursts corresponding to the segment-files with the segment-files' presentation time. Then we train a binary classifier on the extracted VBR pattern of an MP4 file and the (aligned) burst series to tell if the former was generated by the latter.

Alignment is a difficult problem in itself since the extracted 720p MP4 files may not be identical to the actual files used by the service for streaming (which may not even be in 720p). We heuristically tried several values for the alignment of each MP4-capture pair and used neural networks to train a classifier. Our classifier achieved 74% accuracy. This indicates a strong correlation between files and streams of the same video, but is not sufficient for "open-world" identification. Our main approach of using multiple streams of the same video to train the detector achieves much higher accuracy in practice.

B Comparison with Nearest Neighbor

Dubin et al. [20] represent the attacker's measurements of a stream as a set of bursts and use a classifier that maps each such set to the closest training example. Their distance metric is the size of the set intersection. If the intersection is smaller than a threshold, the stream is classified as "unknown."

We implemented this approach and tried it on the Youtube and Netflix datasets, with the train-test split of 0.9-0.1. For the Netflix dataset, which does not contain the "other" class, we used a threshold of 0, i.e., a match is always accepted. For the YouTube dataset, we tried thresholds of 1, 5, and 10.

The nearest neighbor classifier performs very poorly on both datasets. On the Netflix dataset, it attained accuracy of 0.393. On the YouTube dataset, it attained accuracy of 0.624 with threshold 1, 0.05 with threshold 5, and even less with threshold 10. These results show that collisions in burst sizes are simply too rare. Even when the nearest neighbor of a sample is actually found in its correct class, there are fewer than 5 collisions with it.

To further assess the nearest neighbour approach, we applied "bucketing" on all burst sizes, rounding them to a multiple of 10, 100, 1000, and 10000. We found that rounding to a multiple of 1000 is effective, yielding 0.871 and 0.967 accuracy for the Netflix and YouTube datasets, respectively. We call this classifier the *bucket Nearest Neighbor* (bN).

This classifier is still extremely sensitive to noise and thus simple to defeat, e.g., by padding bursts with a few random bytes. This mitigation can be deployed by the streaming service. Moreover, a

[6] There exist tools for downloading MP4 files of content from services such as YouTube and Vimeo.

Dataset	Added Noise?	0bN	1bN	5bN	10bN	CNN
Netflix	No	0.871	X	X	X	**0.959**
	Yes	0.22	X	X	X	**0.909**
YouTube	No	X	0.962	0.967	0.832	**0.991**
	Yes	X	0.851	0.79	0.379	**0.989**

Figure B.1: Comparison of different classifiers. TbN denotes bN (nearest neighbor) with threshold T. CNN denotes our neural network.

classifier that is so sensitive to this mitigation will perform poorly when the attacker's measurements are noisy.

Concretely, we add a random number between 0 and 2% of bytes to each burst size in the dataset. We measure the accuracy of the bN classifier vs. our CNNs. Our classifiers use the total burst series (see Section 5.2). We trained our classifier for 1,400 and 700 epochs for Netflix and YouTube, respectively. For the neural networks, we used a 0.7-0.3 train-test split (as opposed to the 0.9-0.1 for the bN classifiers). Table B.1 summarizes the results.

The KNN classifier of [20] is designed for direct observations of the traffic bursts. We attempted to apply it to the burst estimates as measured from JavaScript. Since these estimates are sums of values returned by `window.performance.now()`, they are measured in milliseconds and in a floating-point representation that captures time at an even finer granularity. Therefore, to make it easier to recognize a (coarse) fingerprint, we used the same approach as above and divided bursts into coarse-grained buckets. We tried 100-second buckets, 10 seconds, seconds, deciseconds, centiseconds, milliseconds, decimilliseconds, centimilliseconds, and microseconds. The KNN classifier of [20] works best at the granularity of 10 seconds, and even then it only attains 0.22 accuracy. We conclude that the approach proposed in [20] does not work when the attack is performed by sandboxed JavaScript code.

C Titles Used in Experiments

Netflix

- "Mad Men" Season 1, episodes 1-10
- "Arrested Development" Season 1, episodes 1-10
- "Narcos" Season 1, episodes 1-10
- "BoJACK Horseman" Season 1, episodes 1-10
- "The Office" Season 1, episodes 1-6, Season 2, episodes 1-4
- "Luke Cage" Season 1, episodes 1-10
- "Louie" Season 3, episodes 1-10
- "Making a Murderer" Season 1, episodes 1-10
- "Stranger Things" Season 1, episodes 1-8
- "Master of None" Season 1, episodes 1-10
- "Parks and Rec" Season 1, episodes 2-3

YouTube

- `https://www.youtube.com/watch?v=lc8804tkoaM`
- `https://www.youtube.com/watch?v=RDfjXj5EGqI`
- `https://www.youtube.com/watch?v=iW-y0Ci5nTI`
- `https://www.youtube.com/watch?v=_clqcSj2rKM`
- `https://www.youtube.com/watch?v=31784aZeJcc`
- `https://www.youtube.com/watch?v=DcJGalE3vn0`
- `https://www.youtube.com/watch?v=uINi-b5Fi1o`
- `https://www.youtube.com/watch?v=bFjrmATIUYU`

- https://www.youtube.com/watch?v=fIOBSUSAikY
- https://www.youtube.com/watch?v=DpdJJN9OYMg
- https://www.youtube.com/watch?v=eyU3bRy2x44
- https://www.youtube.com/watch?v=0fYL_qiDYf0
- https://www.youtube.com/watch?v=Dgwyo6JNTDA
- https://www.youtube.com/watch?v=Z4uN9kh-gdE
- https://www.youtube.com/watch?v=DPeRRWSqPFY
- https://www.youtube.com/watch?v=Th9mfs5eobw
- https://www.youtube.com/watch?v=dUoC-GJ0FQY
- https://www.youtube.com/watch?v=tjhrNKQX29U
- https://www.youtube.com/watch?v=8YkLS95qDjI
- https://www.youtube.com/watch?v=BxKLpArDrC8

Vimeo

- https://vimeo.com/110217114
- https://vimeo.com/111281488
- https://vimeo.com/11671747
- https://vimeo.com/116764246
- https://vimeo.com/120842635
- https://vimeo.com/126371564
- https://vimeo.com/130612876
- https://vimeo.com/138816246
- https://vimeo.com/146489061
- https://vimeo.com/153418170

Amazon The Wire, episodes 3-13 except 10 (these were chosen randomly from the season).