Beauty and the Burst: Remote Identification of Encrypted Video Streams

Roei Schuster, *Tel Aviv University, Cornell Tech;* Vitaly Shmatikov, *Cornell Tech;* Eran Tromer, *Tel Aviv University, Columbia University*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schuster

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Beauty and the Burst: Remote Identification of Encrypted Video Streams

Roei Schuster
Tel Aviv University, Cornell Tech
rs864@cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

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

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 patterns, and 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 does not directly observe the stream, e.g., a JavaScript ad confined in a Web browser on a nearby machine.

1 Introduction

Everything has a fingerprint, and so do encrypted video streams. Transport-layer encryption hides the content but not the network characteristics such as the number of bits transmitted per second. Video streams are known to be bursty [2, 32, 42]. If their traffic patterns are correlated with content, an adversary who can measure them may be able to identify the video being streamed.

There have been several attempts to use traffic analysis to identify encrypted streamed content [1, 11, 43, 44, 46]. Existing techniques, however, generate many false positives, make "closed-world" assumptions (i.e., the adversary must know in advance that the streamed video belongs to a small known set), or are not robust to noise in the network or the adversary's measurements.

Further, prior work assumes that the adversary can directly observe the encrypted video stream either at the network layer (e.g., a malicious Wi-Fi access point) [11] or physical layer (e.g., a Wi-Fi sniffer) [43, 46], or else that the adversary's virtual machine is co-located with the user's virtual machine [1]. 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) but cannot directly observe the encrypted stream.

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. The adversary can measure video fingerprints on his own network and then use them to recognize videos streamed on the target network. 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. 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. In concurrent independent work, Reed and Kranch achieved comparable results for identifying streamed Netflix videos using direct network observations [44] (see Section 11).

Fourth, we demonstrate that video identification based on burst patterns does not require direct access to the stream. Our attack can be performed by a remote attacker who serves JavaScript code (e.g., a malicious Web ad) running under the confinement of the browser's same origin policy, possibly on a different device. For example, if the user is watching Netflix on his TV using a Roku streaming device, his content may be identified by the JavaScript executing on a PC on the same local network. The attack code saturates a shared network link carrying the targeted video stream and uses the result-

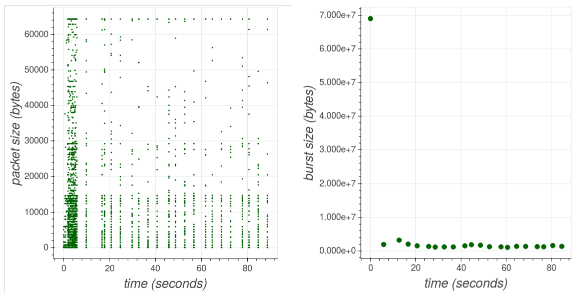


Figure 2.1: Features of a Wireshark capture of Episode 3 of *Mad Men*. The left-hand figure shows packet sizes along the time axis (packet sizes may be larger than Ethernet MTU because of TCP offloading [53])—observe the pattern of buffering followed by the on/off steady state. The right-hand figure shows the size of bursts; the first, largest burst is the size of the buffer.

ing contention to obtain coarse estimates of the stream's traffic rates and identify the video. This attacker is much weaker than malicious ISPs and Wi-Fi access points typically considered in the traffic analysis literature.

In summary, we (1) explain the root causes of burst patterns in encrypted video streams, (2) show how to exploit these patterns for video identification 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 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 [2, 32, 42].

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

Streamed video content is typically segmented at the application layer. 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 [8, 12, 46]. 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 [34, 59] the MPEG-DASH standard [49, 52] 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. The content is stored in segment-files on the server. Each file contains a particular encoding of one segment. When a streaming session is initiated, the server sends to the client a manifest 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.

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 the entire segment finishes downloading long before the client requests another one. Therefore, in a steady-state, on/off stream, burst sizes are correlated with 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 slower scenes. Figure 2.2 shows how the bitrate of an excerpt from the "Iguana vs. Snakes" video [40] in the "Planet Earth" series changes over time (based on an MP4 file downloaded from YouTube). The video starts with an intense chase scene as the iguana is escaping from snakes. In the last 15 seconds, the iguana reaches

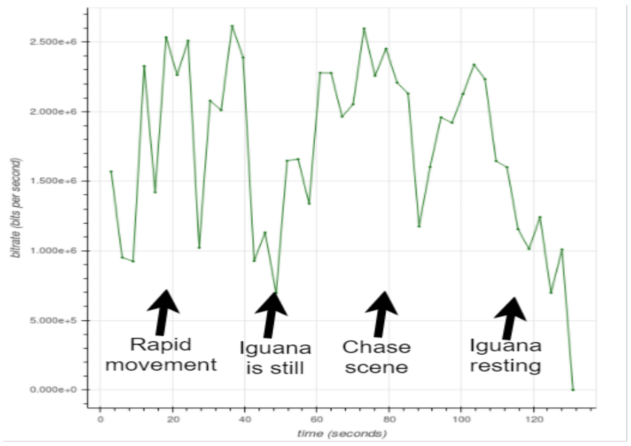


Figure 2.2: Bitrate of the “Iguana vs. Snakes” video.

higher ground and rests 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 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. One of the first HTTPS responses from the YouTube server is an XML Media Presentation Description (MPD), which describes MPEG-DASH segmentation into 5-second segments. The MPD specifies the audio encoding (135 Kilobits per second) and five video encoding options corresponding to different resolutions: 144, 240, 360, 480, and 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 increases. When the segments with a resting iguana are being fetched, it decreases. 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.

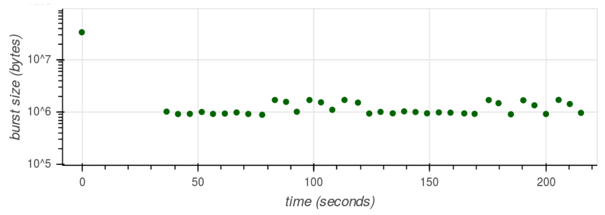


Figure 2.3: Burst sizes when streaming a video with alternating high- and low-bitrate periods. The first, largest burst is the size of the client’s buffer.

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.

Cross-site and cross-device 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 reveal the victim’s traffic patterns. This is a special case of timing side channels in schedulers [16, 25] 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 [51], but it does not prevent the code from using the above timing side channel to measure bursts in a concurrent video stream as long as the stream and the attacker’s own traffic share a network link. The client receiving the stream may be running in a different tab or browser instance on the same machine (a *cross-site* attack) or on a different machine on the same local network (a *cross-device* attack). For example, a smart TV may be streaming a movie while the attacker’s JavaScript is running in a browser on a laptop on the same home network.

3.2 Other attack scenarios

There are several other scenarios where the attacker can indirectly estimate the bitrate 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 [3, 66]. 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) session-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 [15, 17, 23].

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 execute JavaScript in a browser on the same PC), he may be able to estimate traffic via other side channels, such as shared cache or Linux virtual filesystems (sysfs and procfs) [41, 67].

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 file or not.

In this paper, we use machine-learning models as detectors. To generate labeled training data, the attacker streams the video of interest to his own computer and captures the resulting 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: we used 60 seconds per sample for the Netflix on-path attacker, 5-6 minutes per sample for the JavaScript attacker. In our experiments, we targeted the first minutes of the stream, but this approach works for 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 another, target 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 [35, 65] 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 identical segmentation (see Section 7.4).

If the attacker's client and network support the highest-quality encoding, he can also get the service to stream lower-quality encodings by downgrading through the interface of the streaming application, or by 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 [66], 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 not one of the videos for which he has detectors.

5 Experimental Setup

5.1 Targets and attackers

As the streaming client, we used a Chrome browser running in an Ubuntu 14.04 VM on a Windows host with an Intel i7-3720QM CPU. We also experimented with a Roku Premiere streaming device (see Section 9.3).

The clients were connected to a university campus network with over 105 Mbps upload and download bandwidth (measured using [54]). We refer to it as the "training network." For the cross-network experiments in Section 7.4, we also used a campus Wi-Fi network (10 Mbps) and a home Wi-Fi network from a cable ISP (82 Mbps). We refer to them as "test networks."

To evaluate on-path attacks, we assume that the attacker directly observes the target stream as described in Section 5.2. To evaluate off-path attacks, we assume that the attacker executes his JavaScript client code either in the same browser that is receiving the target stream (the *cross-site* attack), or on a machine on the same local network as the device that is receiving the target stream (the *cross-device* attack). In both cases, the attacker's client is communicating with a colluding *attack server*.

In both the cross-site and cross-device scenarios, (1) the attacker's client and the recipient of the target stream are behind a congested home router, while (2) the attack server and the streaming server are outside this router, in different Internet locations. Consequently, the target stream and the attacker's client-server communications share a congested network link. In Section 9, we described our setup for these experiments in more detail.

5.2 Data collection

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

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 spawned a Chrome browser instance and used a service-specific "rewind" procedure so that playback commenced at the beginning of the content. For videos with an initial title sequence, this (non-unique) sequence is downloaded as part of the initial buffering; the bursts in the on-off phase correspond to the segments of unique content.

We captured the network traffic of each streaming session for a certain duration (see below) using Wireshark's `tshark` [60]. 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 a Chrome failure or network glitch. The resulting captures contained very few bytes and we discarded them.

Feature extraction. From each capture, we kept only the TCP flow with the greatest amount of bits and extracted the time series of the following *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, we aggregated the series into 0.25-second chunks by averaging over 0.25-second intervals.

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 I (we used $I = 0.5$). 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 the midpoint between the beginning and the end of the point sequence that forms the burst. The value of the burst is the sum of the values of points in the sequence. We aggregate bursts series by summing into 0.25-second chunks.

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 and captured each of the 20 selected titles 100 times, and each of the 3,558 titles from the automated crawl once. Encoding for YouTube videos varies and bitrate can be less variable than for Netflix; also, the content is sometimes preceded by an ad. Therefore, we took 4.5-minute Wireshark captures and cropped the captured streaming flows to 3 minutes. For 2 of the 20 titles, the ad was so long that the capture of the actual content was shorter than 3 minutes. We discarded these and only used the remaining 18 titles, with 3-minute content captures for each.

We also downloaded actual 720p MP4 file video files (as opposed to their network streams) for the 3,558 titles from the crawl, using the SAVEFROM.NET Web tool. These files were used for measuring the uniqueness of burst patterns, not for identification experiments.

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.

Storage. After feature extraction, the data saved for our attack experiments totals 1.2GB for Netflix, 2.3GB for YouTube, and about 0.5GB each for Vimeo and Amazon.

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 [4] to process our 3,558 YouTube videos (see Section 5.2) for standardized streaming, i.e., divide them into time segments and create the manifests. We opted for 5-second segments, which matches our observations of both Netflix and YouTube and is close to a recent recommendation [10]. We believe that the encoding parameters of these videos are representative of other YouTube videos. The MPEG-DASH client-server interaction induced by our simulated server 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 traces. 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.

For the theoretical analysis in this section, we use a very simple fingerprinting algorithm. For any $v = (v_1, \ldots v_k) \in \mathbb{R}^k$, define $\alpha(v) \equiv (v_1, \ldots v_k, v_2 - v_1, \ldots v_k - v_{k-1})$. Intuitively, $\alpha(v)$ accounts for both the absolute magnitudes of segment sizes and their variability pattern.

During training, the attacker acquires n training traces $TS = \{t_1, \ldots t_n\}$ drawn from T^m. Let $s^m = \text{mean}(TS)$, the element-wise average over TS. Training produces $\alpha(s^m)$, which is the attacker's *fingerprint* of m.

During the attack, the attacker is given the victim's trace $t \in \mathbb{R}^k$ and computes its *traceprint*, $\alpha(t)$. The attacker 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. To compute the recall, or true positive rate, of this attack, 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 attacker-measured traceprint and the fingerprint of the underlying video) 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 away 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[1] $t \in \mathbb{R}^{40}$ into 4 contiguous blocks of length 10 and apply the union bound on the probabilities of error in each block and the difference elements in α, i.e., $\left|(t_i - t_j) - (s_i^m - s_j^m)\right|$ for $(i, j) \in (11, 10), (21, 20), (31, 30)$. For each of the 7 elements of α, the error is bounded by $B/7$ with probability $\geq 1 - 10^{-12}$. Total error is thus bounded by B with very high probability, $\Pr_{t \leftarrow T^m} [\|\alpha(t) - \alpha(s^m)\|_1 \leq B] \geq 1 - 7 \cdot \left(10^{-12}\right) \geq 1 - 10^{-11}$, implying very high recall.

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

Attacker's precision. Even if the distance between the attacker-measured "traceprint" and the video's fingerprint is small, the attacker may still misclassify the video if its fingerprint is close to another one. We show that for almost 20% of the videos in our YouTube dataset, such mistake 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 video pair $m \in V, m' \in D \cup V$ such that $m \neq m'$, $\left\|\alpha(z^m) - \alpha(z^{m'})\right\|_1 \leq 2B$. Then our attacker could mistake m for m' even if m's traceprint is B-close to the fingerprint (as must be the case with high probability). There are no such collisions in our dataset.

To estimate the attacker's precision, we need to assume that s^m, the series of average burst sizes used to compute the fingerprint, 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 $\alpha(s^m), \alpha(s^{m'})$ are $2B$-close in the L_1 norm. Since with probability 10^{-11} a traceprint $\alpha(t)$ (of a video m with variable segment size) is B-close to the correct fingerprint $\alpha(s^m)$ (by the recall bound above), the probability that an attacker mistakes t's video for another one 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 of 3,558 in our dataset. The attacker can easily check whether a particular video satisfies this criterion.

While our dataset is small in comparison to the entire YouTube, the extremely low error rate and complete absence of collisions indicate that the attack should 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).

In the following sections, we develop a 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., packet timing). Moreover, the simple classifier based on L_1 distance 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 [29] is a branch of machine learning based on multi-layer artificial deep neural networks (DNNs). DNNs have proved very effective for signal recognition tasks such as speech transcription [19], image segmentation [14], image classification [28], 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 subsequent (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.

DNNs are good at capturing high-level concepts that are easy for humans to agree on but hard to express formally. In our case, we use DNNs to capture traffic-level commonalities of the streaming sessions of a given title, even in the presence of some traffic variations among these sessions. Further, neural networks are flexible and can leverage information from the low-level features, such as packet lengths, as well as sequences of burst sizes (as estimated from encrypted traffic). As input, they can use any time series that characterizes the stream. We exploit this in both on-path and off-path attack scenarios.

Convolutional Neural Networks (CNNs) [9] are deep neural networks whose lower layers apply the same linear transformation on many windows of the input data.

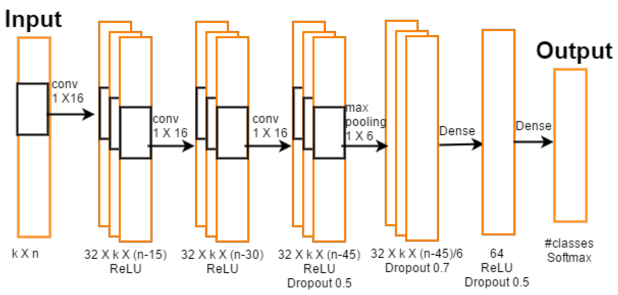


Figure 7.1: Our CNN architecture. k denotes the number of feature types taken. n is the recording time in seconds divided by the time-series sampling rate (0.25).

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 setting, where the network events corresponding to each DASH burst occur in close temporal proximity.

We use supervised training on a corpus that consists of traffic measurements labeled with their correct class, i.e., the identity of the corresponding video. Training involves multiple *epochs*. During each epoch, an optimization procedure processes a batch of training data and adjusts the parameters in the functions computed by the layers so as to minimize the error between the correct classification and the output of the classifier. Learning is successful only if (1) the classifier reduces the training error, and (2) the reduced error rate generalizes to test samples, i.e., inputs that the classifier was not trained on.

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 [26] 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.

7.3 Classification results

We trained a separate classifier 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 test samples.

The YouTube classifier is remarkably accurate. 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.

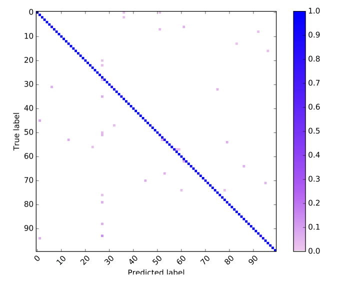
Minimizing false positives. The output of the last, softmax layer of the neural network is traditionally interpreted as a vector of probabilities. The classifier's prediction is the class with the highest probability. We can use this probability as a confidence measure.

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 confidence is below the threshold, we intentionally classify the input as "other" regardless of the class chosen by the classifier.

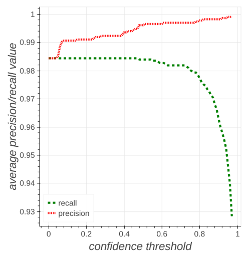
Figure 7.3b shows the precision and recall of the classifier for various values of the confidence threshold. Precision and recall are calculated by aggregating the false positives and false negatives of all classes except "other". Without any decrease in recall, we can achieve a false positive rate of 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 than 0.0005, or precision of 0.9995, with just 1 false positive out of 2224 matches.

YouTube 1/18 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 the confidence threshold. Even when the threshold is 0 (equivalent to simply taking *argmax* of the classifier's output), the false nega-

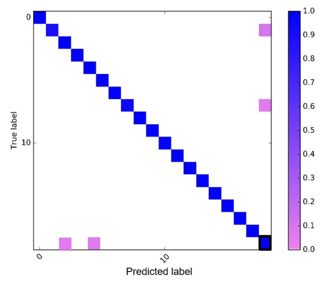


(a) Confusion matrix. The entries off the diagonal are misclassifications. Color in cell i, j denotes the number of samples of class i classified as j.

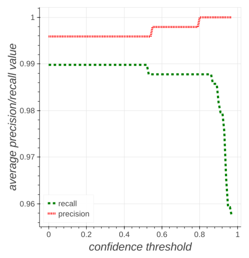


(b) Precision vs. recall.

Figure 7.3: Netflix 1/100 classifier.



(a) Confusion matrix. The entries off the diagonal are misclassifications. The bottom row and rightmost column are of the "other" class.



(b) Precision vs. recall.

Figure 7.4: YouTube 1/18 + "other" classifier.

tive 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 test accuracy after training. Training on multiple features was 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.

Dataset	TIME	EPOCHS	PLEN$_{\text{IN}}$	PLEN$_{\text{OUT}}$	PLEN	BPS$_{\text{IN}}$	BPS$_{\text{OUT}}$	BPS	BURSTS	BURSTS$_{\text{IN}}$	BURSTS$_{\text{OUT}}$	PPS$_{\text{IN}}$	PPS$_{\text{OUT}}$	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	0.980	0.938	0.984	0.984	**0.986**	0.916	0.958	0.924	0.940

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 test 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.

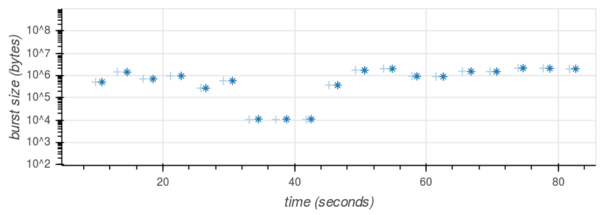


Figure 7.5: Burst sizes of streamed “Reservoir Dogs”. The two captures were made on a campus network (+) and a home network ($*$).

The attacker can still collect training data 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).

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.

7.5 Possible improvements

Our classifiers attain very high accuracy but can benefit from some potential improvements.

First, our Netflix classifier was trained on just 60-second captures, equivalent to only about 45 seconds of steady-state bursts after the (less discriminative) buffering period. It may be possible to train an even more powerful classifier using 90-second captures.

Second, our relatively simple classifiers are slightly under-fitted. More expressive classifiers (e.g., with more hidden layers) suffer from over-fitting, but it may be solved with more data, e.g., 1000 captures per video.

Finally, the low base rate potentially motivates the use of *detection cascades* [56] consisting of a series of classifiers, each of which is more complex (with a larger input feature space and more hidden layer activations) than the previous one. During training, the $(i+1)^{\text{th}}$ classifier is trained using only the samples accepted (possibly falsely) by the i^{th} classifier. A cascade thus accepts only the inputs that are accepted by all of its classifiers and is efficient to train because most inputs are rejected by the simple lower-level classifiers. Cascades have demonstrated almost human-level accuracy for complex tasks with low base rate such as face detection [64].

8 Bayesian Detection Rate

In Sections 6 and 7, we showed detectors with very low false positive rates. However, the attacker's false detection rate is not the detector's raw 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 the victim is streaming m (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' 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 being streamed, is characterized by an extremely low *base rate*, i.e., probability $P(M)$ that the video actually corresponds to any of the attacker's detectors. In this setting, when the attacker's recall is sufficiently high, BDR is dominated by the false positive rate.[2]

[2]For example, suppose the recall is $\Pr(A|M) = 1$, false positive rate is $\Pr(A|\neg M) = \frac{1}{1000}$, and the victim streams 100,000 videos sequentially. If the attacker has a detector for one of them (i.e., the base rate is $\frac{1}{100000}$), he would get roughly 100 false matches before the true match.

We now analyze the detectors from Sections 6 and 7 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 in our set D were likewise sampled from $\psi^{\hat{D}}$ (i.e., by sampling videos according to their popularity on the service). Let $V \subseteq D$ the the videos in D with variable segment size. Assume the attacker has a detector for some $m \in V$.

Let $t \leftarrow T^{m'}$ be an observed trace. If the detector matches but $m' \neq m$, then either $\left\|\alpha(s^{m'}) - \alpha(t)\right\|_1 \geq B$, or $\left\|\alpha(s^{m}) - \alpha(s^{m'})\right\|_1 \leq 2B$. The probability of the former is low because the recall is very high, $> 1 - 10^{-11}$. Let p_{COL} denote the probability of the latter event, corresponding to a collision between two videos.

If $p_{\mathsf{COL}} \geq \frac{2}{10^6}$, then we are likely to observe a collision in our dataset D. Under the simplifying assumption that collisions in D are independent events,[3] with overwhelming probability $1 - (1 - p_{\mathsf{COL}})^{(|D|-|V|)|V|+|V|^2/2} > 0.986$ there exist $m_V \in V, m_D \in D$ such that $m_V \neq m_D$ and $\|\alpha(s^{m_D}) - \alpha(s^{m_V})\|_1 \leq 2B$. Since we did not observe any such collisions in 2,162,297 pairwise tests over D, it is likely that $p_{\mathsf{COL}} \leq \frac{2}{10^6}$.

In this case, assuming the open-world base rate is $\frac{2}{10^6}$, 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 the 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 $\Pr(A|\neg M)$ and $\Pr(A|M)$ into the BDR formula and expect to get a good estimation, since the distribution that this classifier was trained on—without samples from the catchall "other" class—is fundamentally different from the distribution of videos that might be streamed by the victim.

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

When collisions do not occur, our classifier, tuned for precision over recall, performs very well and misclassifies only 1 out of 2,224 test samples. 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 220,176 possibilities.

9 Off-path Attacks

9.1 Measurement with JavaScript

Consider a remote attacker who has a restricted foothold in the victim's network. For example, he controls an ad embedded in some webpage visited by the victim. An ad may include JavaScript code executing in the victim's browser, but because this code may come from a questionable source with strong commercial interest in users' data (including their viewing habits), it is confined—both by the main browser sandbox, which prevents it from issuing arbitrary requests to the OS, and by the same origin policy [51], which prevents it from accessing the content that belongs to other Web origins. In particular, even if the victim is streaming a 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 the attacker's 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 video being streamed by the victim. Since Internet links usually have bounded bandwidth, this means that the attacker's JavaScript and the video stream share a limited resource. JavaScript can send and receive arbitrary amounts of data to and 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 communication between the attacker's JavaScript code and its own server. The attacker can then estimate how much traffic is flowing over the link by measuring these delays. This leaks information about the content streamed from a different origin by the same browser (a **cross-site attack**, see Fig-

[3]This assumption is an approximation. It could have been strongly violated, e.g., if all collisions are due to a small set Z of videos, each of which collides with many other videos: if 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. If the fingerprints of videos in Z are close to those of many other videos, then the latter videos also have fingerprints that are geometrically close to each other and are thus likely to collide in D.

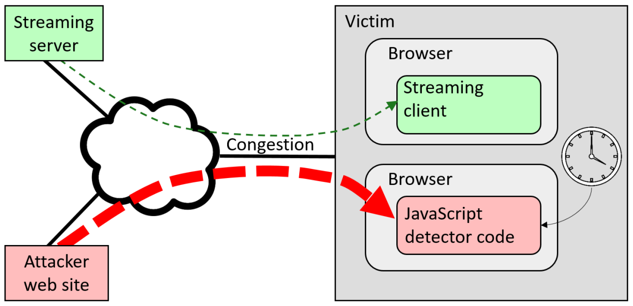


Figure 9.1: Cross-site attack.

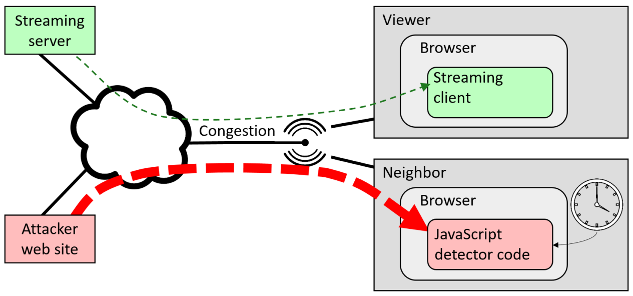


Figure 9.2: Cross-device attack.

ure 9.1), or even by a different device on the same local network (a **cross-device attack**, see Figure 9.2).

9.2 Simulating the attack

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 estimate contention from other traffic on the shared link.

Network. To simulate cross-site and cross-device attacks, we run two browser windows concurrently, one streaming the selected video, the other executing a JavaScript attack client. In the cross-site setup, they run on the same virtual machine. In the cross-device setup, they run on different machines. Both machines are on a home network, *victim-LAN*, behind a traffic-throttling router that simulates a bandwidth-limited connection. In the cross-site setup, we simulate *victim-LAN* and the router with VMware Workstation. In the cross-device setup, we use an actual home router. In both cases, the router is connected to the Internet via a university LAN. The attack server is on the same LAN. All traffic between *victim-LAN* (which includes the streaming client and the attack JavaScript client) and the Internet (which includes the streaming server and the attack server) thus flows through a bandwidth-constrained router.

Data. We used 10 Netflix titles: 5 episodes from the first season of "Mad Men" and 5 arbitrary other titles. We streamed each title 100 times and used a JavaScript client to indirectly measure the traffic as described above. We used 5-minute captures in the cross-site experiment and 6-minute captures in the cross-device experiment.

Cross-site attack. The attacked machine was an Ubuntu 14.04 VM, with a simulated 45 Mbps (5.625 MBps) down/upstream bandwidth (capped by VMware Workstation). 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.[4]

From the $\{X_n\}$ vector of message arrival times measured by the attacker's client, we compute the vector of message delays $Y = (0) \| ((X_2, \ldots X_n) - (X_1, \ldots X_{n-1}))$ and filter the X, Y time series for delays that exceed 8ms. We then compute the burst series as in Section 5.2 with 0.25-second intervals and filter out the bursts whose sizes are below 80, producing a *delay bursts* time series. To create uniformly sized vectors, we aggregate this series by averaging into 0.25-second chunks.

Cross-device attack. 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 network. 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 sending an 8KB message every 1.5ms, about 300 KBps short of saturating the network link.

In this experiment, we smoothed the time series of the delay measurements by averaging over 0.1-second intervals, filtered it for delays $y > 2.1$ms, computed the burst series with 0.5-second intervals, and filtered out all bursts whose sizes were below 10. To create uniformly sized vectors, we chunked it into 0.1-second intervals.

Classifier. We used a variant of the classifier from Section 7.2 that we found less prone to overfitting on the noisier, longer samples in this attack. Between the last max-pooling layer and the first fully-connected layer, we added another convolution layer, with kernel size 7, followed by a max pooling layer (both with ReLU activations). We applied 0.7 dropout after every hidden layer. All other convolution-layer dimensions were changed to 1x12 and pooling-layer dimensions to 1x2. We used 16 filters for all hidden layers instead of 32. Finally, we used Adadelta instead of the Adam optimizer.

[4] A portion of messages is queued at the server, taking up to 500MB of memory. In the cross-device attack, we calibrated the transmission rate in a different way, alleviating this.

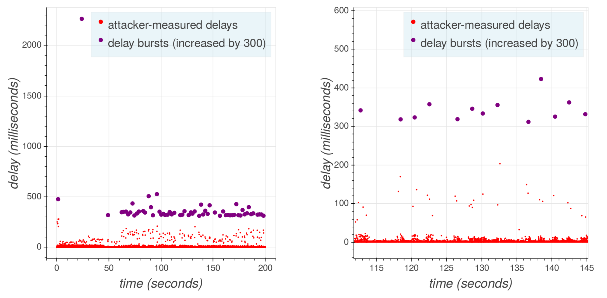


Figure 9.5: Cross-device attack on a Roku streamer. On the left is the global view, including initial buffering. On the right is the local view during steady-state streaming. Bursts cause a visible increase in delays observed on the neighbor machine.

9.3 Results

In all of our experiments, attacks were imperceptible to the user and did not affect the viewing quality.

Cross-site attack. Figure 9.3a shows that bursts in the video stream are very visible in the measurements performed by the JavaScript client. Fig. 9.3b shows that the delay bursts series is strongly correlated with the bursts of the actual stream. Our 1/10 Netflix classifier attains 0.937 accuracy. As in Section 7, we can adjust our confidence threshold to reduce false positives at the cost of reducing recall (see Fig. 9.3c). By accepting 0.793 recall, we obtain precision of 1.

Cross-device attack. The timing of the messages observed by the detector code on the neighbor device exhibits clear patterns corresponding to the stream received by the viewer device. Figure 9.4a shows that bursts in the stream during the steady state cause delays in the messages received by the neighbor. Figure 9.4b shows that delay bursts are correlated with the size of bursts in the stream (which, in turn, reflect segment sizes). Our classifier performs well, with 0.965 accuracy. By accepting 0.933 recall, we obtain precision of 0.997.

Cross-device attack on a Roku streamer. Many users watch streaming video content on a smart TV or a dedicated streaming device connected to a TV. To investigate the feasibility of our attack in this scenario, we used the cross-device attack setup from Section 9.2 except that the viewer was a Roku Premiere streaming device (a very popular brand), connected to the Internet via Wi-Fi.

The bursts corresponding to video segments are clearly observable from a neighbor machine. Figure 9.5 shows the attacker-measured delays while Roku is streaming Episode 1 of "Mad Men." They exhibit the expected pattern of a large burst followed by smaller ones in steady intervals, each lasting a few seconds.

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 off-path attack, the attacker's server sends large amounts of traffic to congest a shared network link and his JavaScript client measures arrival times in the victim's browser. To create congestion, the server needs a high-bandwidth connection to the victim's network. Therefore, success of the off-path attack using a specific server may depend on the victim's location and ISP.

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. Specifically, in adaptive streaming, encoding quality can be dynamically downgraded or upgraded in response to changing network conditions. In this paper, we did not evaluate a scenario where the victim is experiencing erratic network conditions causing frequent switches between encodings.

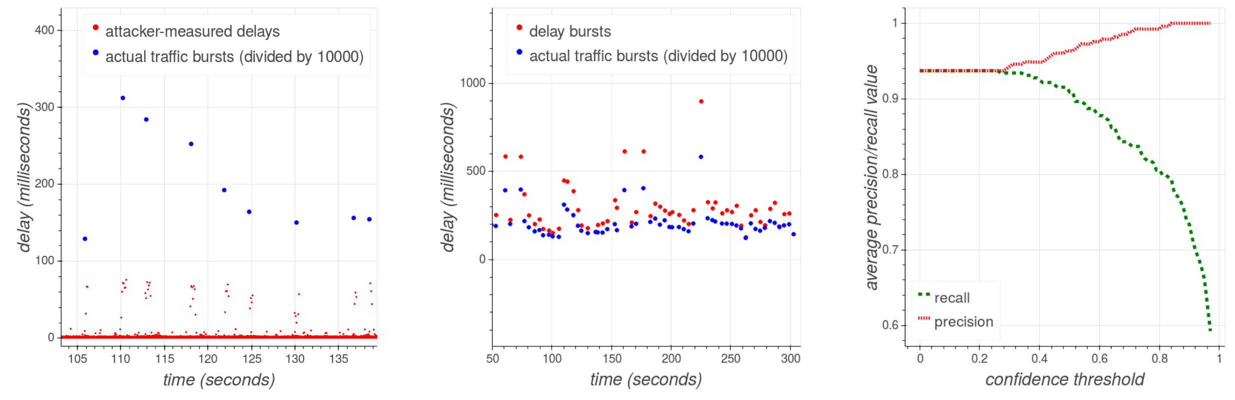
Our techniques aim to identify standard, unmodified streaming video (e.g., Netflix movies). They are not designed to resist evasion. If the user or service re-encodes the video (e.g., at a different resolution), the attacker's previously trained detectors will no longer work.

Our techniques can be automated and deployed on a reasonably large scale to detect hundreds or thousands of titles in an "open-world" setting, without assuming a priori that the video belongs to small known set. Scaling beyond that is likely to be expensive. Data collection is the main bottleneck because training detectors requires the attacker to stream the same title multiple times.

11 Related Work

11.1 Exploiting VBR leaks

Fine-grained video. Saponas et al. [46] observed that encrypted, VBR-encoded videos leak information about their content. To create a "signature" of a video, they

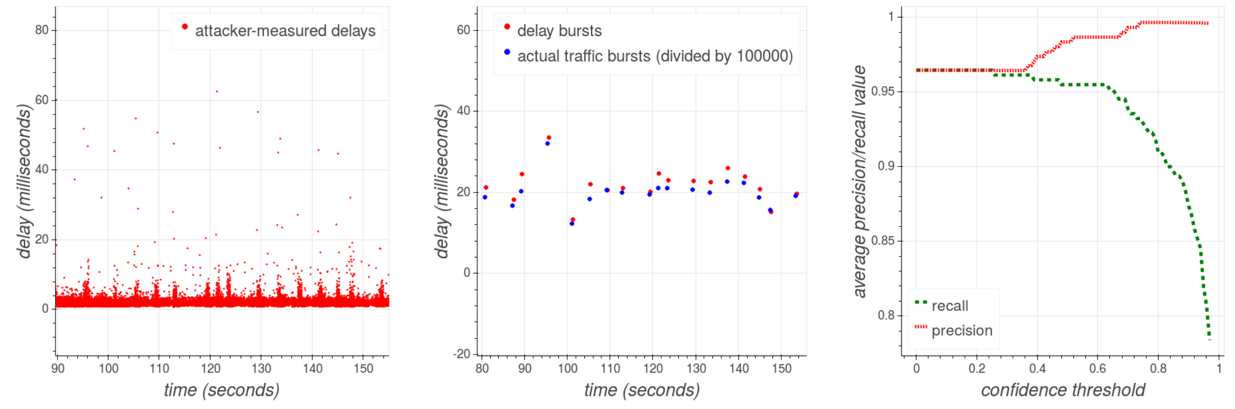


(a) Actual bursts size vs. measurements from JavaScript in a different origin.

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

(c) Precision vs. recall of our classifier

Figure 9.3: Cross-site attack.



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

(b) Delay bursts vs. actual traffic bursts. Traffic bursts are divided by 100,000 for presentation.

(c) Precision vs. recall of our classifier

Figure 9.4: Cross-device attack.

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. [30] focus on detecting re-encoded content. They apply a wavelet transform to the time series of frame sizes and cross-correlate the wavelet coefficient series of the observed traffic with those of a reference content file. In [31], Liu et al. use aggregated traffic throughput traces (as opposed to frame-size time series) and report 1% false positive rate and 90% recall rate.

These methods operate on time series resembling, and close to the granularity of, the sizes of individual frames. DFTs and wavelet transforms capture short-term variations due to changes of picture and long-term variations due to changes of scene. In our setting, the observable features are bursts 4–6 seconds (120–180 frames) apart.

Even though these methods rely on fine-grained measurements, their false positive rates are prohibitively high for "open-world" identification (with a low base rate, 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., performed by sandboxed JavaScript) are noisy and coarse-grained.

Dubin et al. [11] suggest using the (unordered) set of segment sizes as a title fingerprint. 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.

Reed and Klimovski [43] implement a Wi-Fi sniffing attack and suggest an approach based on Pearson correlation for identifying Netflix streams. In a preliminary evaluation, they report correctly identifying, given 50 possible titles, 24 out of 25 streaming sessions. Concurrently and independently from our work, Reed and Kranch [44] scale this approach by fingerprinting the entire Netflix title selection. They assume an on-path attacker who can observe TCP-layer traffic. This approach has not been evaluated in an off-path setting, where the attacker has only noisy side-channel measurements, nor for any streaming services other than Netflix.

Mass fingerprinting in [44] relies on the metadata sent by Netflix to the client at an early stage of the streaming process, namely the .ismv file headers that contain all segment sizes for all possible encodings of the title. They are sent in the clear, while the video content is DRM-encrypted. It is not clear how the approach of [44] would work if these headers were DRM-protected, too.

VoIP. Wright et al. showed that VBR leakage in encrypted VoIP communication can be used to identify the speaker's language [62] and detect phrases [61]. Their detector is a Hidden Markov Model trained to identify a specific phrase. White et al. [58] 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 [21, 37, 45].

Our network congestion attack works because traffic-flow scheduling policies for a shared internet link are leaky. Kadloor, Gong, et al. [16, 24, 25] studied the tradeoffs between delays, fairness, and privacy in scheduling policies on shared resources. Kadloor et al. [23] 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 [15, 17]. This attack is 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 router that discards unsolicited packets from outside the network (as many modern routers do by default).

Agarwal et al. [1] 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 concurrent video streams has never been empirically demonstrated before. This is a particularly dangerous scenario because untrusted JavaScript code from sources who have commercial interest in users' viewing habits is ubiquitous on the Web.

Timing attacks have a long history in computer security [6, 50]. Felten and Schneider [13] observed that JavaScript can infer information from the timing of cross-origin requests; Bortz and Boneh [5] demonstrated several timing-related Web attacks; Van Goethem et al. [55] proposed timing techniques that tolerate network noise and server-side mitigations. Oren et al. [36] used JavaScript timing mechanisms for a cache attack. Kohlbrenner and Shacham [27] 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 [7, 8, 15, 18, 20, 39, 48, 57]. Juarez et al. [22] argue that most of these efforts make unrealistic assumptions and fail to cope with the base rate fallacy. Panchenko et al. [38] 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 [8, 33], as well as Web sources of video traffic [47]. Prior work also includes mitigations [63] and counter-mitigations [12].

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 (which is primarily done to maximize quality of experience) reduces the granularity of the leak but does not prevent video fingerprinting.

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, 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, in general, an affine function of presentation time). Solving the problem requires a different buffering regime. Client-side-only changes are easier to deploy than changes to segmentation on the server, but devising such a regime is non-trivial even if we allow changes to both client side and server side.

For example, consider a *variable-size buffer* that fetches equally-sized segments every X seconds (where X is fixed). This requires a balance between increasing the fetching rate (lest the buffer runs out in the middle of long high-action scenes) and increasing the initial buffering time (for robustness to network conditions while also accounting for sudden buffer depletion due to high-bitrate content). Both factors would directly degrade user experience and network efficiency.

Network congestion side channel. The congestion attack requires big, frequent server-to-client messages that may appear anomalous and thus recognizable at the network level. Detection and prevention mechanisms can be placed at the router, network, OS, or browser. A more sophisticated attack implementation may be able to use benign-looking traffic to circumvent such mechanisms. Fuzzy-time sandbox solutions such as [27] would not entirely prevent our attack: the JavaScript client can still send packets to congest the uplink, yet timing measurements can be performed by a 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 off-path 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 videos being streamed by the user on the same or different device.

Acknowledgements. Roei Schuster and Eran Tromer are members of the Check Point Institute for Information Security. This work was supported by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); Defense Advanced Research Project Agency (DARPA) and Army Research Office (ARO) under Contract W911NF-15-C-0236; Google Faculty Research Awards; Israeli Ministry of Science and Technology; Israeli Centers of Research Excellence I-CORE program (center 4/11); Leona M. & Harry B. Helmsley Charitable Trust; and National Science Foundation awards CCF-1423306, CNS-1445424 and CNS-1612872. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of ARO, DARPA, NSF, the U.S. Government or other sponsors.

References

[1] 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 *CANS 2016*.

[2] Pablo Ameigeiras, Juan J Ramos-Munoz, Jorge Navarro-Ortiz, and Juan M Lopez-Soler. Analysis and modelling of YouTube traffic. *ETT 2012*.

[3] John S Atkinson, O Adetoye, Miguel Rio, John E Mitchell, and George Matich. Your WiFi is leaking: Inferring user behaviour, encryption irrelevant. In *WCNC 2013*.

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

[5] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW 2007*.

[6] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[7] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *CCS 2012*.

[8] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in Web applications: A reality today, a challenge tomorrow. In *S&P 2010*.

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

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

[11] 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*.

[12] 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 *S&P 2012*.

[13] Edward W Felten and Michael A Schneider. Timing attacks on Web privacy. In *CCS 2000*.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR 2014*.

[15] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *PETS 2012*.

[16] Xun Gong and Negar Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *Biological Cybernetics*, 2016.

[17] Xun Gong, Negar Kiyavash, and Nikita Borisov. Fingerprinting websites using remote traffic analysis. In *CCS 2010*.

[18] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial Naïve-Bayes classifier. In *CCSW 2009*.

[19] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 2012*.

[20] Andrew Hintz. Fingerprinting websites using traffic analysis. In *PETS 2002*.

[21] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1992.

[22] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *CCS 2014*.

[23] Sachin Kadloor, Xun Gong, Negar Kiyavash, Tolga Tezcan, and Nikita Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *ICC 2010*.

[24] Sachin Kadloor and Negar Kiyavash. Delay optimal policies offer very little privacy. In *INFOCOM 2013*.

[25] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *Biological Cybernetics*, 2016.

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

[27] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security 2016*.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS 2012*.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[30] 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 *GLOBECOM 2008*.

[31] Yali Liu, Ahmad-Reza Sadeghi, Dipak Ghosal, and Biswanath Mukherjee. Video streaming forensic–content identification with traffic snooping. In *ISC 2010*.

[32] Jim Martin, Yunhui Fu, Nicholas Wourms, and Terry Shaw. Characterizing Netflix bandwidth consumption. In *CCNC 2013*.

[33] 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 *PETS 2014*.

[34] 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.

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

[36] 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 *CCS 2015*.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*.

[38] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. Website fingerprinting at Internet scale. In *NDSS 2016*.

[39] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES 2011*.

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

[41] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *CCS 2012*.

[42] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network characteristics of video streaming traffic. In *CONEXT 2011*.

[43] Andrew Reed and Benjamin Klimkowski. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11n connections. In *CCNC 2016*.

[44] Andrew Reed and Michael Kranch. Identifying HTTPS-protected Netflix videos in real-time. In *CODASPY 2017*.

[45] 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 *CCS 2009*.

[46] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security 2007*.

[47] Yan Shi and Subir Biswas. Protocol-independent identification of encrypted video traffic sources using traffic analysis. In *ICC 2016*.

[48] Yi Shi and Kanta Matsuura. Fingerprinting attack on the Tor anonymity system. In *ICICS 2009*.

[49] Iraj Sodagar. The MPEG-DASH standard for mul-

timedia streaming over the Internet. *IEEE MultiMedia*, 2011.

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

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

[52] Thomas Stockhammer. Dynamic adaptive streaming over HTTP: Standards and design principles. In *Multimedia Systems 2011*.

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

[54] Testmy: Web-based bandwidth test. http://testmy.net/. Accessed: 2017-01-16.

[55] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern Web. In *CCS 2015*.

[56] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR 2001*.

[57] Tao Wang and Ian Goldberg. Improved website fingerprinting on Tor. In *WPES 2013*.

[58] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *S&P 2011*.

[59] 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.

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

[61] 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 *S&P 2008*.

[62] 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 2007*.

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

[64] Shuo Yang, Ping Luo, Chen-Change Loy, and Xiaoou Tang. WIDER FACE: A face detection benchmark. In *CVPR 2016*.

[65] 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.

[66] Fan Zhang, Wenbo He, Xue Liu, and Patrick G Bridges. Inferring users' online activities through traffic analysis. In *WiSec 2011*.

[67] 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 *CCS 2013*.

A Streams vs. MP4 Files

Since the cause of the leak is the DASH standard, it would be nice to compute detectors directly from video files[5] instead of streaming each video multiple times.

This approach faces several challenges. First, the attacker must infer the exact segmentation parameters, such as segment duration and minimal buffer time, and how they change with respect to file encoding, size, bitrate, view count, etc. Each service has many combinations of these parameters. Furthermore, they change over time but changes may not apply to the already-segmented files. Second, this approach does 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. First, we have to align the stream with the file, i.e., match traffic bursts corresponding to segment-files to 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 because the extracted 720p MP4 files may not be identical to the actual files used by the streaming service (which may not even be in 720p). We heuristically tried several values to align each MP4-capture pair and used neural networks to train a classifier. Our classifier achieved 74% accuracy. This indicates a strong correlation between the files and the streams of the same video, but it 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. [11] 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. If the size of the set intersection is smaller than a threshold for all examples, the stream is classified as "unknown."

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

Dataset	Added noise?	0B	1B	5B	10B	CNN
Netflix	No	0.871	*X*	*X*	*X*	**0.959**
	Yes	0.220	*X*	*X*	*X*	**0.909**
YouTube	No	*X*	0.962	0.967	0.832	**0.991**
	Yes	*X*	0.851	0.790	0.379	**0.989**

Table 1: *T*B (bucketed nearest neighbor classifier with threshold *T*) vs. CNN (our neural network).

We implemented and tried this approach for the Netflix dataset (which does not contain the "other" class, so we used a threshold of 0, i.e., a match is always accepted) and the YouTube dataset, with thresholds of 1, 5, and 10. The test-train split was 0.9-0.1.

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

To further assess this approach, we "bucketed" all burst sizes by rounding them to a multiple of 10, 100, 1000, and 10000. 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* classifier (B).

This classifier is still very sensitive to noise and will perform poorly if the attacker's measurements are noisy or if the streaming service deliberately pads bursts with a few random bytes. We added a random number of bytes between 0 and 2% to each burst size in the dataset and measured the accuracy of the B classifier vs. our CNN-based classifiers, which use the total burst series (see Section 5.2) and are trained for 1,400 and 700 epochs on the Netflix and YouTube data, respectively. We used the 0.7-0.3 train-test split for the CNNs (vs. 0.9-0.1 split for the B classifiers). Table 1 summarizes the results.

The KNN classifier of [11] is designed for direct observations of the streaming traffic. We attempted to apply it to the burst estimates as measured from JavaScript. Because 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 [11] works best at the granularity of 10 seconds, and even then it only attains 0.22 accuracy. We conclude that the approach proposed in [11] does not work for an off-path attack.

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 Recreation" 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-yOCi5nTI`
- `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: 10 episodes chosen arbitrarily from Season 1 of "The Wire": 3, 4, 5, 6, 7, 8, 9, 11, 12, and 13.