MiBao: A Video Processing Middlebox

Siyuan He

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-18
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-18.html

May 1, 2017

Acknowledgement

I want to thank my research advisor Professor Randy H. Katz for guiding me to through my Master's study. I wish to thank Professor Sylvia Ratnasamy for her valuable advice during the development of this project. I wish to thank my collaborators Zehao Wu and Syed Saif Nizam who worked with me on this project. I want to thank my colleagues Kaifei Chen, Jack Kolb, Gabe Fierro, and Michael Andersen for their advice and helping me proof read the thesis. This work is supported in part by the National Science Foundation under grant CPS-1239552 (SDB), AMPLab, and the Siebel Scholars Foundation.

MiBao: A Video Processing Middlebox

by

Siyuan He

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Randy H. Katz, Chair
Professor Sylvia Ratnasamy

Summer 2016

MiBao: A Video Processing Middlebox

Abstract

MiBao: A Video Processing Middlebox

by

Siyuan He

Master of Science in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

Network video streaming such as video hosting websites and connected cameras are increasingly popular. These applications bring new challenges to network administrators, such as leakage of privacy information and exposure to inappropriate content. When network administrators do not have control over end-hosts or processing video at individual end-hosts results in duplicate computation or a large configuration burden, they can mitigate such risks by inspecting video stream on the wire through Deep Packet Inspection (DPI). While the rise of Software-Defined Networking (SDN) and middleboxes brings new functionalities to existing networks, they are usually limited to fixed field processing.

In this thesis, we summarize the performance, architectural and format characteristics of network video streaming, design and implement the MiBao Video Processing Middlebox, and evaluate its overhead, effectiveness and scalability. Through allowing MiBao to vary the parameters of a face detection algorithm with respect to load and leveraging on GPU acceleration, MiBao can perform on-the-wire face detection and blur for up to four 1080p video streams at 25 FPS with less than 100ms playback delay using an AWS g2.2xlarge EC2 instance. As a proof-of-concept, MiBao shows that with GPU and tunable algorithms, middleboxes can meet video packets' delivery deadlines while performing complex computations.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my research advisor Professor Randy H. Katz for guiding me to through my Master's study. I wish to thank Professor Sylvia Ratnasamy for her valuable advice during the development of this project. I wish to thank my collaborators Zehao Wu and Syed Saif Nizam who worked with me on this project. I want to thank my colleagues Kaifei Chen, Jack Kolb, Gabe Fierro, and Michael Andersen for their advice and helping me proof read the thesis. This work is supported in part by the National Science Foundation under grant CPS-1239552 (SDB), AMPLab, and the Siebel Scholars Foundation.

Chapter 1

Introduction

With increasing bandwidth, video streaming through the Internet has become pervasive. Streaming sites such as YouTube readily provide millions of users with a variety of video content[15]. Network enabled cameras such as the Nest Cam help households monitor their property through the Internet [8]. While bringing convenience to people, these applications also impose security and privacy risks such as leakage of private information through network enabled security cameras [28] and exposure to inappropriate content from video streaming websites [30, 54, 65]. To reduce the impact of such risks, video streams need to be inspected and filtered for inappropriate content. Such demand creates new challenges for local network administrators.

Similar to operations on web content, video inspection and filtering could be done at three points in the network: source, destination and on the wire. At the source, video streaming websites such as YouTube often remove inappropriate video when flagged by staff, users or authorities [36]. Connected cameras such as the Nest Cam try to protect users' privacy through having a "Home" mode that allows users to turn off the camera when they do not want to be filmed [5]. At the destination, users can install local Web Content-Filtering Systems to filter content using URL blocking, keyword filtering and even artificial neural networks (ANN) [41]. However, both methods have feasibility and overhead issues. On one hand, the source of a video stream may not cooperate with users' demands on content inspection and filtering. While YouTube moderates its content, many websites do not; while the Nest Cam has a convenient ON/OFF switch, other connected cameras may not. On the other hand, filtering content at local machines requires redundant configuration at individual end-hosts, especially for large enterprise networks with uniform policies. Even worse, such filtering may not be feasible in mobile and low power devices where processing power is scarce. Given the limitations of end-host inspection and filtering, performing these operations on the wire through Deep Packet Inspection (DPI) becomes an advantageous solution.

However, while the rise of Software-Defined Networking (SDN) and middleboxes adds new functionalities to existing networks at fast pace, few works have explored the solution for in-stream video processing. In the past, network administrators performed DPI on non-video network traffic to protect security and privacy of local networks. Researchers

have developed packet filters [44], firewalls at different layers, Intrusion Detection Systems (IDS) [56], and exfiltration prevention devices [61]. Most of these appliances are limited to fixed field processing and textual pattern matching, and are not capable of performing complex operations on visual semantic content in video streams.

The top challenge for in-stream video processing is managing intensive computations at scale while meeting video packets' delivery deadlines and users' functional requirements. Firstly, as video processing tasks often involve unconventional middlebox operations on a large block of application layer payload, such as encoding, decoding, signal processing, and applying computer vision algorithms, their behavior on a network appliance was not well-explored, let alone how the tasks behave at scale. Moreover, video content, especially real-time captured ones, is only useful to receivers when delivered within a short deadline. This characteristic limits the amount of time a video packet can stay in a middlebox to between 10 ms to 1 s, depending on the application. Lastly, while meeting these computational constraints, the middlebox also needs to fulfill the functional requirements set by its user, such as accuracy for a face detection middlebox. Due to these reasons, researchers in the past did not have much confidence in building such middleboxes [18].

This paper explores the feasibility of building a video processing middlebox and the set of requirements it must achieve. Recently, given the efficiency gain in GPU based computer vision and video processing [53], we believe that it is feasible to build such a middlebox with commodity hardware. To identify its requirements, we first summarize the timing, architectural and format characteristics of network video streaming. Using this information, we derive the latency and throughput requirements for a video processing middlebox. We then design and implement MiBao, a middlebox that detects and blurs human faces in video streams sent by connected cameras in order to protect users' privacy.

Starting with a baseline implementation that hardly meets the requirements, we optimize MiBao by relieving bottlenecks at encoding and face detection. We implement a tunable context-aware face detection algorithm and experiment with its parameters to profile the trade-offs between performance and accuracy. By exposing these parameters to the middlebox controller, enabling GPU acceleration, and pipelining stages, MiBao can process up to four 1080p video streams at 25 FPS with less than 100 ms playback delay overhead, less than 1% frame loss and more than 60% accuracy on an Amazon Web Service (AWS) g2.2xlarge instance. We further optimize MiBao to work with multiple GPUs, enabling support for up to twelve such video streams at the same accuracy on a g2.8xlarge instance.

One thing this thesis does not explore is security. While there are many video encryption algorithms that encrypt video streams at different stages, most of them still rely on certificate-based methods such as SSL for key distribution [42]. If a middlebox is able to catch the encryption key and knows which encryption algorithm is adopted by the video stream, it can decrypt the video stream with ease. There are existing solutions such as *SSL Bumping* through which a middlebox uses a forged certificate to perform a man in the middle "attack" on the SSL key exchange [38, 35]. In our context, since the network administrator is trusted by the local network users, this "attack" is legitimate. Moreover, as many networks have already deployed such technology, providing decryption for video will not incur

additional configuration overhead. However, as video decryption and encryption may incur extra computational cost, it may affect the performance of our middlebox. We leave this part for future researchers to explore.

Chapter 2

Network Video Streaming

2.1 Processing Time Constraints

In network video streaming, two delays affect the viewing experience: the *inter-frame delay* and the *playback delay*. The inter-frame delay is the inverse of frame rate (e.g. 40 ms for 25 FPS). A multi-threaded processing pipeline will not affect this delay if a single stage's processing time is less than the original inter-frame delay. The playback delay is the sum of propagation and transmission delays from all stages, and an application's requirement for this delay creates processing time constraints that a video processing middlebox must meet, which can be categorized into three types as shown in Table 2.1.

Table 2.1: Video Streaming Timing Constraints

Type	**Timeliness**	**Interaction**	**Tolerable Playback Delay**	**Processing Time Constraint**
1	Stored Video	View-Only	10s	1s
2	Real-Time	View-Only	1s	100ms
3	Real-Time	Two-Way	100ms	10ms

Type 1, Stored Video, View Only: This type refers to real-time transmission of stored video [70]. Video is played while being downloaded from a remote repository such as YouTube and Netflix or through a distributed protocol such as BitTorrent. Users are not sensitive to playback delay, but care more about playback smoothness that is given by consistent inter-frame delays. Thus the maximum playback delay can be as large as tens of seconds.

Type 2, Real-Time, View Only: This type refers to real-time generated video stream that is transmitted one-way only. It includes all types of network enabled cameras and live television broadcasts. Users are more sensitive to playback delay as a video's content has timeliness value. However, as video source do not expect immediate feedback from the viewer, users' tolerance on the playback delay can be as long as several seconds. For example,

an IPTV application's delay could be up to 6.5 seconds [40]. For connected cameras, the common delay is about 1-10 seconds [29].

Type 3, Real-Time, Two-Way Interaction: This type refers to real-time video streams that involve back and forth interactions between the source and the destination. It includes all kinds of video conferencing applications in which the party at the video source expects immediate feedback from the party at the destination. Users are very sensitive to playback delay and can only tolerate it at the scale of human reaction, about 100 ms [20].

Therefore, when building video processing middleboxes, the additional delay due to middlebox processing should not exceed the original application's delay by too much. If we set our limit at 10% of the tolerable playback delay for the original application, the processing time constraints for each type would be 1 s, 100 ms, and 10 ms respectively as shown in Table 2.1.

2.2 Streaming Format

Figure 2.1: Video Streaming Pipeline

When video is streamed over a network, it needs to be encoded to a smaller size, muxed into a container format, packetized as a media streaming protocol payload and sent via appropriate transport layer protocols as illustrated in Figure 2.1. When being played at the destination, a video packet needs to go through the reverse procedures including depacketizing, demuxing and decoding before being displayed at an end-host. Each stage has a variety of choices that an application can implement. Table 2.2 presents a summary of our discovery on the choices available for each stage.

Table 2.2: Available Choices on Each Video Streaming Stage

Coding Format	H.264, H.265, WMV, VPX, MPEG-2
Container Format	MPEG-TS, MP4, FLV, ASF, 3GP, QuickTime, RMVB, or None
Media Streaming Protocol	RTP, Apple HLS, Adobe HDS, RTMP, BitTorrent Live, Skype

Coding Formats

Without encoding, a constant rate raw video stream includes a series of images called frames. For raw video frames:

$$\text{Frame Size} = \text{Color Depths} \times \text{Height} \times \text{Width} \times \text{Channels}$$

Thus an 8-bit RGB 1920×1080 frame takes 6.22 MB. At 25 frames per second (FPS), it results in a data rate of 1.24 Gbit/s or 155 MB/s. Even with today's network bandwidth that averages to about 14.2 Mbps in 2015Q4 for the U.S. [23], such a rate is hardly achievable. However, as a raw video stream involves duplicate and redundant information across pixels and frames, various encoding algorithms were invented to reduce its size. For example, H.264, a common coding format adopted by new applications, can reduce data rate of the above video stream to 4.92 Mbit/s at normal quality [52]. Besides H.264, other protocols such as VPX, WMV, MPEG-2, and H.265 also exist and are used by a variety of applications.

Container Formats

A container describes a schema that combines video, audio, subtitles and metadata into a bitstream, allowing them to be stored or streamed over network. For streaming applications, common container formats include MPEG Transport Stream (MPEG-TS), Flash Video (FLV), QuickTime, RMVB, and MP4. These container formats differ by their supported coding formats, ability to enforce copyright protection, etc. The choice of container format largely depends on the choice of multimedia streaming protocol. However, for applications using Real-time Transport Protocol (RTP) [58], no container is needed as each channel is streamed separately using different RTP connections.

Multimedia Streaming Protocols

When video is streamed over a network, a multimedia streaming protocol defines the procedures to convert video into network packets. A multimedia streaming protocol can be built on top of existing application protocols such as HTTP Live Streaming (HLS) [50], HTTP Dynamic Streaming (HDS) [6] or BitTorrent Live Streaming [64]. It could be based on TCP such as Silverlight [7], or UDP such as the previously mentioned RTP and Real Time Messaging Protocol (RTMP) [51]. For Type 2 and 3 real-time applications, UDP is usually chosen in favor of TCP as re-transmission of video packets voids their timeliness value. Such choice limits the pool of multimedia streaming protocols to ones like RTP, RTMP and the Skype Protocol [21]. For example, the Nest Cam streams H.264 encoded video using a variation of RTP over UDP [26].

Chapter 3

System Overview

3.1 Proposed Functionality

As a proof of concept, we aimed to build an exfiltration prevention middlebox that detects and blurs faces in video streams sent by connected cameras in a local network. Upon success, MiBao should be able to apply the same algorithm on videos sent by other applications in Type 1 or 2 with a small tweak in the packet filter and protocol daemons. We did not target Type 3 video streams as we wish to leave it for future work to explore.

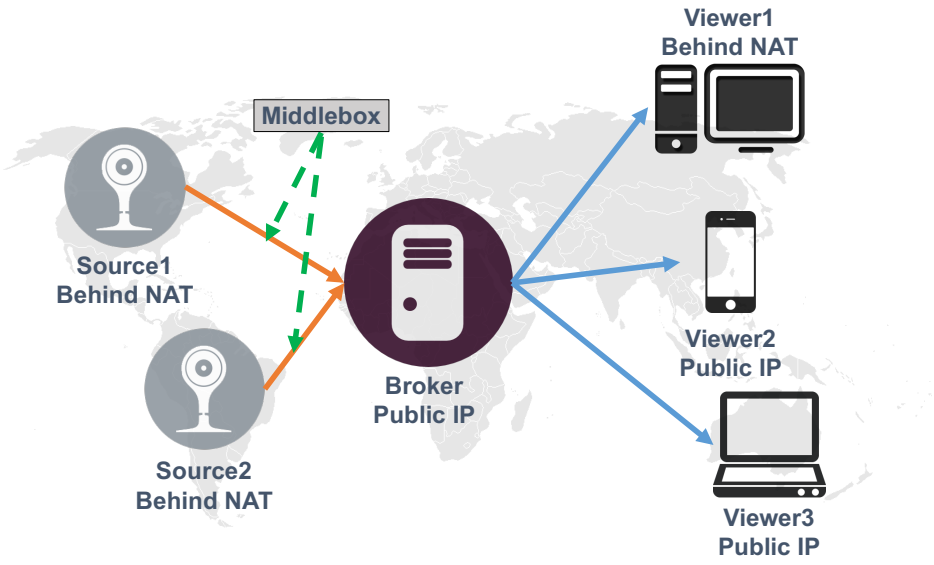

Figure 3.1: Example Streaming Application

In particular, we chose the Nest Cam as the target video streaming application. It encodes video in the H.264 format and uses a variation of the RTP protocol over UDP. To simplify stream extraction, instead of reverse engineering Nest Cam's streaming protocol, we built a real-time streaming application using RFC 6184 [69] as its media streaming protocol.

Table 3.1: Middlebox Specification

Item	Description
Application Protocol	RTP
Transport Protocol	UDP
Coding Format	H264
Resolution	Up to 1920x1080
Number of Streams	Up to 4
Frame Rate (FPS)	Up to 25
Operation	Detect and blur ANY face on the video stream
Additional Latency	Less than 100 ms
Accuracy	Above 60%

In our application, a connected camera sends a video-only stream to an online broker that immediately forwards the video stream to viewers as shown in Figure 3.1. Since we use RFC 6184, no container is required. In the future, audio streams could be buffered alongside video streams to maintain their synchronization with each other.

Since our target application falls in Type 2, MiBao should add no more than 100 ms playback delay to the video stream. The face detection should perform with a reasonable accuracy with a small number of false positives and false negatives. As a network appliance, MiBao should be able to process multiple streams at the same time with minimum frame loss and as much accuracy as possible. Given the above discussion about various goals and constraints, we summarize MiBao's specifications in Table 3.1.

3.2 Architectural Design

As shown in Figure 3.2, between the camera source and destination, MiBao inserts four components into the network: *Packet Filter*, *Gateway Daemon*, *Middlebox Daemon*, and *Video Processor*. Overall, the system is designed to be transparent to the video streaming application so that no additional configuration is required.

Packet Filter

Our application transmits RTP-H.264 video streams through UDP port 5004 because RFC 3551 [57] specifies 5004 and 5005 as the default data and control port pairs. Since our main focus is on video processing instead of video stream extraction, we simplified the Packet Filter by extracting any UDP packets with destination port number 5004. The Packet Filter should reside on a local gateway where all local network traffic must pass through. Extracted packets are then forwarded to the Gateway Daemon that typically runs on the same machine.

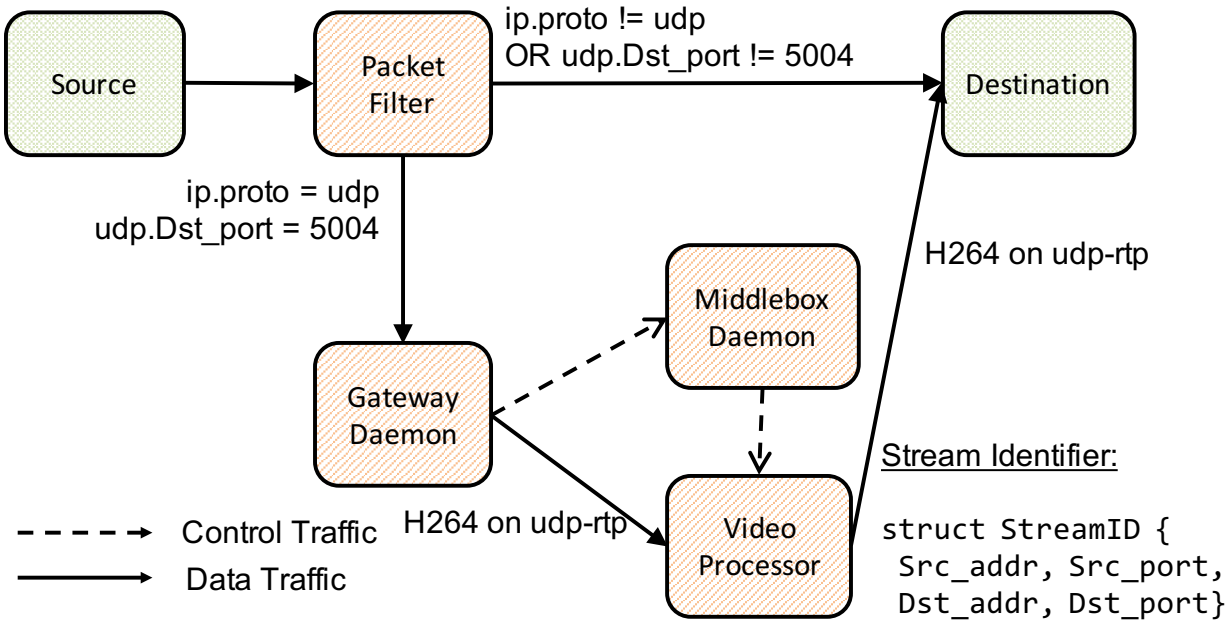

Figure 3.2: Proposed Architecture

Gateway and Middlebox Daemon

The Gateway and Middlebox Daemons implement a simple protocol that manages streams inspected by MiBao. They are two separate processes because we envision that the Video Processor may need to run on a separate machine that has more computational power than the local gateway. With this design, it is possible to put the middlebox on the cloud, which was proven to have the benefits of lower maintenance cost and more flexible provisioning [60].

In detail, when the Gateway Daemon receives packets from the Packet Filter, it extracts a 4-tuple from the UDP packets including: `source address`, `source port`, `destination address` and `destination port`. This 4-tuple serves as a `Stream ID` to identify RTP streams going out from the local network. When a new stream is identified by the Gateway Daemon, it contacts the Middlebox Daemon using the `Stream ID` to request a Video Processor instance. The Middlebox Daemon will set up a new instance of Video Processor to process and forward video to the stream's original destination, and return the instance's network address and port number to the Gateway Daemon. Both the Gateway and Middlebox daemons keep track of running Video Processor instances using in-memory hash tables mapping from Stream ID to Video Processor instance information.

For the Middlebox Daemon, if a Video Processor instance has already been started for a Stream ID, the Middlebox Daemon will only return the information of an existing instance. If a Video Processor instance has quit for a Stream ID, the Middlebox Daemon will remove it from the hash table and restart it at the next request for the same Stream ID.

For the Gateway Daemon, if a Stream ID has been mapped to a Video Processor instance,

it would only perform a hash table lookup and forward the packet to the Video Processor. When the hash table entry is first created upon a successful request from the Middlebox Daemon, it puts an expiration timestamp, typically 10 minutes from current time, in the entry. Once the timestamp expires and it is still receiving packets with the same Stream ID, it will request Video Processor information from the Middlebox Daemon again to update the expiration timestamp. Otherwise, it will remove the entry once timestamp expires. This is not the ideal packet forwarding path since forwarding packets to the Gateway Daemon incur an extra hop. Ideally, the Gateway Daemon should be able to modify the Packet Filter to forward video streams directly to the Video Processor using Software Defined Networking technologies such as OpenFlow [45]. However, as both the Packet Filter and the Gateway Daemon run on the same machine and the overall traffic is small in our test system, this imperfection does not affect our experimental results by large. Moreover, although UDP packets for video streams are large in size, they are sent at a low rate comparable to the video frame rate. Therefore, the hash table lookup will not happen very frequently and will not add too large an overhead to MiBao for Type 1 and 2 applications.

For a video stream, the first few packets will be dropped by the Gateway Daemon until it has a Video Processor instance for that stream. Therefore, for viewers, the video stream will start with a large delay, but once started, the playback delay is comparable to the original application without a middlebox.

Video Processor

The Video Processor is a separate process running on the same or different machine as the Middlebox Daemon. As shown by the shaded boxes in Figure 3.3, the Video Processor is a nine-stage pipeline. It performs UDP receive, RTP depacketize, H.264 decode, and color space conversion to recover raw frame for processing. After processing, it restores the video stream to its original format using the reverse procedures and sends packets to the stream's original destination as instructed by the Middlebox Daemon. All stages operate on the data in-place as much as possible, minimizing memory copy overhead.

Figure 3.3: Video Processing Pipeline Configuration

3.3 Face Detection Algorithm

Among object detection algorithms, Haar feature-based cascade classifiers is an effective and efficient algorithm [67]. Haar features are patterns of pixels that can be extracted from a region of an image. By running a window across the entire image, an algorithm can recognize tens of thousands of Haar features. In the training stage, feature detection is done on a large

set of true positive and true negative examples, producing a set of Haar features that can be used to identify humans' faces. In the detection stage, the same feature detection is done on the image and a comparison against the trained model will expose locations with high probability for faces.

In MiBao, we use the Viola-Jones face detection framework [68]. Instead of detecting over 160 thousand Haar features on the input frame, the Viola-Jones framework first reduces the large feature set down to about 6000 and uses each of them in a weak classifier. It then introduces the concept of Cascade of Classifiers by grouping the 6000 weak classifiers into 38 stages with 1, 10, 25, 25, and 50 features in the first five stages. For each window on the image, the framework applies classifiers from each stage consecutively and the window is discarded if not all classifiers are positive. This results in a robust and faster face detector that can be applied on a single machine.

Chapter 4

Video Processor Optimization

With the baseline design in Figure 3.3, MiBao hardly meets its specifications. The first implementation could only process a single video stream at 7 FPS with more than one minute playback delay overhead. The following steps were taken to reduce latency and improve the throughput of the system.

4.1 Multi-threaded Pipeline

When the pipeline was first implemented, we observed an increasing length of delay as time progressed. Moreover, the middlebox crashed after running for an extended amount of time. By looking at memory utilization, we identified the issue to be UDP receiving buffer overflow. This was because down-stream packet processing rate was much slower than up-stream packet arrival rate. As a result, the output video became increasingly delayed as later arrived packets are buffered for longer time. The system eventually crashed because of buffer overflow.

To resolve this problem, we pipelined the Video Processor using multiple threads with queue buffers inserted between adjacent stages as shown in Figure 3.3. To determine locations of the queues, we measured processing time for each pipeline stage and discovered that the majority of processing time was taken by Processing and Encoding. Therefore, we put two queues at both the up-stream and down-stream sides of the Processing stage and configured the queues to drop the oldest packet when full.

With just this modification, the delay became consistent and the memory footprint became stable. However, the video stream is still significantly delayed because the queue is always full and every frame has to wait to pass through the entire queue. Moreover, as frames are being dropped, the resultant video stream experiences jitter.

4.2 GPU H.264 Encoding

MiBao spends the second most amount of time on the Encode stage when using CPU-based x264 codec [14]. If these resources could be freed, the Processing stage can use a larger portion of CPU time and hence process frames faster. In fact, for H.264 encoding, many graphics cards have embedded hardware encoders that use very little CPU resources. Since our test machine has an NVIDIA graphics card, we adopted the NVIDIA Video Codec SDK [9] to achieve this optimization.

4.3 Enable CUDA

A further optimization is moving the Viola-Jones face detection and blur operations to the GPU. A typical NVIDIA GPU has thousands of CUDA cores that run SIMD tasks in parallel. As mentioned in Section 3.3, the Viola-Jones face detection applies the same cascade of classifiers on large number of windows on the image. Similarly, the box filter and Gaussian filter convolve the same kernel for the entire image. Both tasks could be easily parallelized by the GPU and enjoy a reduction in processing time and CPU usage [53].

4.4 Tunable Algorithm for Middlebox

Even with pipelining and GPU accelerators, processing still takes too much time. The middlebox requires more than 50 ms to process a single 1920×1080 frame. For a 25 fps video stream, an individual stage cannot take more than 40 ms. Further optimizations were taken to bring processing time into the 40 ms limit.

Tunable Context Aware Algorithm

Although the Viola-Jones face detection is quite efficient, it does not store state between consecutive video frames. Given face locations in a frame and assuming the color distributions on these faces vary little between frames, the face locations in the following M frames can be obtained using CAMShift (Continuously Adaptive Meanshift), a fast algorithm that tracks motions of objects using color distribution [25]. Moreover, if no face is detected in a frame, we assume that no face will appear in the next T frames and therefore remain idle in those T frames. With T and M as tunable parameters, we implemented a context-aware algorithm similar to [11], as shown in Listing 4.1. As CAMShift and no-op involve less computation than Viola-Jones face detection, larger T or M will result in faster processing. However, as both assumptions become less valid when no face detection is performed in a long time, larger T or M will reduce detection accuracy, creating a trade-off between performance and accuracy.

```
1  t, m = 0, 0
2  while(has_frame()):
3    frame = get_frame()
4    if t > 0:
5      t -= 1
6    elif m > 0:
7      m -= 1
8      faces = camshift_track(frame)
9      blur_faces(frame, faces)
10   else:
11     faces = vj_detect_face(frame)
12     if len(faces) > 0:
13       m = M
14       init_camshift(faces)
15       blur_faces(frame, faces)
16     else:
17       t = T
18   send_frame(frame)
```

Listing 4.1: Pseudocode for tunable context-aware face detection algorithm using Viola-Jones and CAMShift

Expose Algorithm Parameters to Middlebox

Our next step is to expose both parameters to the middlebox controller. Without exposing them, the middlebox runs a rigid algorithm that is configured for a maximum load level, L_{max}, as shown in Figure 4.1. When load is below L_{max}, the middlebox runs inefficiently with unused resources. When load is above L_{max}, the middlebox starts dropping frames. If we assign zero accuracy to a lost frame, the average accuracy will fall sharply through a cliff. However, by exposing algorithm parameters to the middlebox controller, the middlebox can select parameters based on load level, trading accuracy with performance at run time.

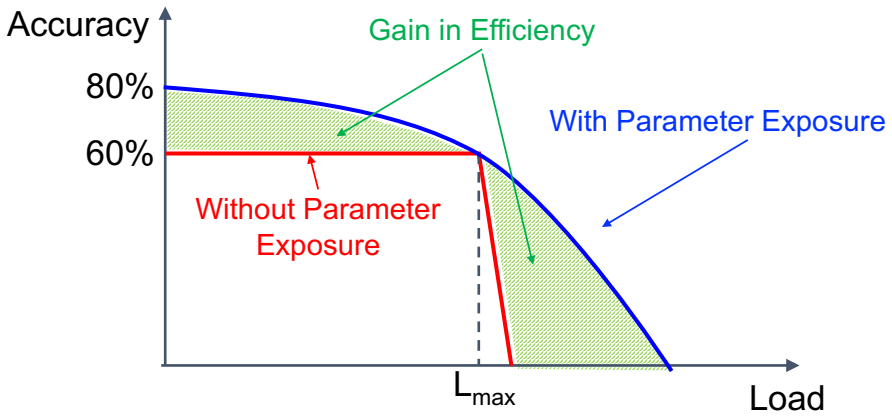

Figure 4.1: Algorithm Behavior

Chapter 5

Implementation

5.1 Packet Filter

The Packet Filter is built to run on a commodity machine running the Linux operating system. The machine should consist of at least two network interfaces. One interface, WAN, connects to the external network. Another interface, NET, connects to the local network. As shown in Figure 5.1, for our experimental system, a TUN virtual interface is created to act as a rendezvous point. NAT, DHCP and DNS are set up using the `iptables` and `dnsmasq` packages between the WAN and TUN interfaces, providing both gateway and router functionality. A Click Software Router [39] is set up between the TUN and NET interface to provide the packet filter service. We used the Click Software Router to implement the packet filter instead of `iptables` because we envision more sophisticated stream extraction beyond UDP port number in the future.

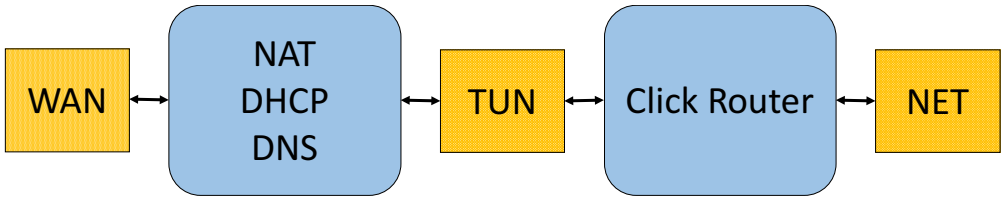

Figure 5.1: Gateway Configuration

5.2 Gateway & Middlebox Daemon

The Gateway and Middlebox Daemons are implemented using Go as it provides an easy-to-use thread programming model. The Middlebox Daemon sets up an HTTP server that listens to requests in JSON format from the Gateway Daemon.

5.3 Video Processor

The Video Processor is implemented using the GStreamer Framework [63]. Among a number of available open source video processing frameworks including GStreamer, FFMpeg [3], and VLC [13], we chose GStreamer because it is modular and uses GLib [4] to manage all low level interactions with the operating system such as memory allocation, thread scheduling and cross platform compatibility. Using such a framework, we can focus our implementation on video processing itself instead of logistics. In light of deploying middleboxes on the cloud, the Video Processor is packed in a Docker container so that the Middlebox Daemon can set up the Video Processor on multiple machines and limit its resources when necessary.

In our GStreamer video processing pipeline, each stage in Figure 3.3 corresponds to a GStreamer element. It includes sources that generate or obtain media streams, filters that transform media streams, muxers and demuxers that combine or separate multimedia streams, queues that store media content temporarily and sinks that store or send media to a location. Data handles are transferred among elements by pointers unless specially configured for copying, minimizing memory copy overhead as much as possible. It uses GLib's GObject system to implement these elements and Glib's mainloop is used to handle scheduling among various GObjects. All such elements are dynamically loaded to the main program at run-time, providing a separation between the pipeline management and data processing.

In particular, we created a special element called `mibaofaceblur` to implement our face detection and blur algorithm using OpenCV [10], an open-source computer vision library. In detail, this element receives a raw 8-bit RGB image buffer from up-stream, operates on the buffer in-place, and pushes the processed buffer's handler down-stream. It first converts the RGB frame to grayscale and then runs face detection using `cv::CascadeClassifier` class, the library's implementation of the Viola-Jones face detection framework. The class was initialized only once using OpenCV's trained Haar Cascade model for frontal faces: `haarcascade_frontalface_default.xml`. After detection, all faces are blurred twice using a box filter and a Gaussian filter before the buffer is pushed.

Chapter 6

Evaluation

6.1 Experiment Deployment

We conducted experiments to characterize MiBao's performance. The application itself is built using the architecture from Figure 3.1. For the online **Broker**, we used a t2.micro instance from the AWS us-east-1a service zone [1]. The Broker runs a VLC [13] server that receives video stream through RTP and distributes it over HTTP. For the local video **Source**, we constructed a GStreamer pipeline that captures real-time video from a webcam, encodes it in H.264, and sends it as RTP payload over UDP. As a GStreamer pipeline, it runs across multiple platforms including the Raspberry Pi and OS X. The **Viewer** can be a machine that runs a VLC player. For our experiment, it is a Macbook Pro running OS X.

The middlebox components were deployed on two machines: a **Gateway** and a **Middlebox**. For the Gateway, we used an x86 machine running Ubuntu 14.04 LTS with Core 2 Duo 7400 at 2.80 GHz and 4 GB RAM. It has a 1GbE interface (WAN) connected to an external network and a 100MbE interface (NET) connected to an internal network. Both the Packet Filter and the Gateway Daemon run on this machine. For the Middlebox, we deployed it on the cloud where more GPU resources are available. Specifically, we used a g2.2xlarge instance from the AWS us-west-2b service zone, which is an x86 Machine running Ubuntu 14.04 LTS with 8-Core Xeon E5-2670 at 2.60 GHz with 15 GB RAM. It has a 1GbE interface and an NVIDIA Grid K520 graphics card. Both the Middlebox Daemon and the Video Processor run on this machine.

As shown in Figure 3.2, the Source is connected to the internal network and has Internet access through the Gateway. Source, Gateway and Viewer are all located within Soda Hall at UC Berkeley. The Middlebox is located in Oregon. The online broker is located in Virginia. As both middlebox and destination are on the same cloud service provider with high-bandwidth, low-latency links between them, the amount of additional delay is minimum, even though they are located at different service zones. Moreover, as mentioned in Chapter 1, middlebox security is not the main focus of this experiment and previous research has proven that a middlebox on the cloud can be both efficient and secure [60].

6.2 Test Data

As many experiments require running the Video Processor against the same video stream multiple times, we played stored video at capturing rate to simulate the content of a video surveillance camera. In particular, we took video data from the MOT16 Multiple Object Tracking Benchmark [46], using the #08 and #12 video samples as our test examples. From individual frames provided by the dataset, we reconstructed them into 25 FPS 1920×1080 H.264 video streams at 4992 kbps.

6.3 Playback Delay

As discussed in Section 3.1, the playback delay of our application, which is the end-to-end delay from the Source to the Viewer, should be smaller than 100 ms. To measure this delay, a frame needs to carry a microsecond timestamp that is used to compute its end-to-end delay at the Viewer. For measurement to be accurate, the Source and the Viewer must be on the same machine that uses the same hardware clock.

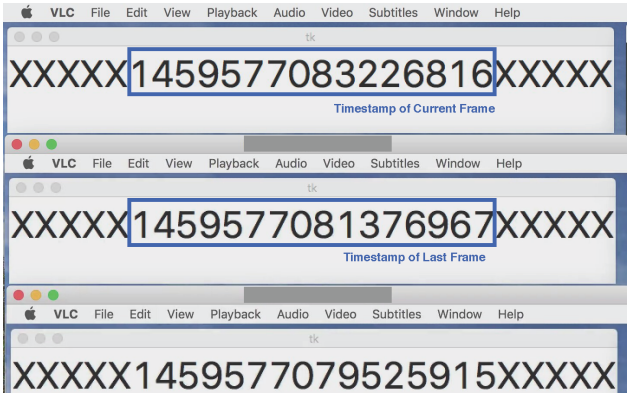

Figure 6.1: Screenshot for Playback Delay Measurement

Since neither the RTP H.264 payload format nor the VLC streaming server provides a reliable way to preserve metadata, the only way a video stream can carry a timestamp is by embedding it in the video frames [33, 24]. To achieve that, we implemented a test application that streams local screenshots to the remote broker and plays the video on the local machine at 1440×900 resolution and 25 FPS. A microsecond timer is shown on the local machine so that the final screenshot has nested windows of microsecond timers as shown in Figure 6.1. The difference between the microsecond timer on the outermost layer and second-to-outermost layer is therefore the true playback delay for the last frame. We then configure our application to save a screenshot to local disk every second and later parse all screenshots using the Tesseract OCR Engine [62] to obtain microsecond precision playback delays. We have considered using a barcode to reduce the OCR error rate. However, since the barcode method requires image rendering that takes more than 30 ms on our machine and we wanted to obtain microsecond level measurement precision, the method was discarded.

Using a wired connection, we conducted experiments for each step of the optimization in Chapter 4 for 10 minutes each. As shown in Figure 6.2, the CDF of playback delay shows that although our streaming application itself has a relatively large playback delay of about 1883 ms on average, the optimized middlebox only adds 51 ms to that delay, far less than the 100 ms target. While the baseline middlebox adds a large delay of 18770 ms, pipelining, GPU encoding, CUDA and tunable algorithm reduce the delay by 16105 ms, 640 ms, 380 ms, 1594 ms respectively. Without the multi-threaded pipeline, buffer overflow causes the CDF for baseline system to have a large variation and a long tail on the right. Pipelining solves this problem and brings playback delay to a stable distribution. Besides pipelining, the tunable algorithm has the second largest impact on reducing the playback delay, although the result could be very specific to face detection and depend on the complexity of test video.

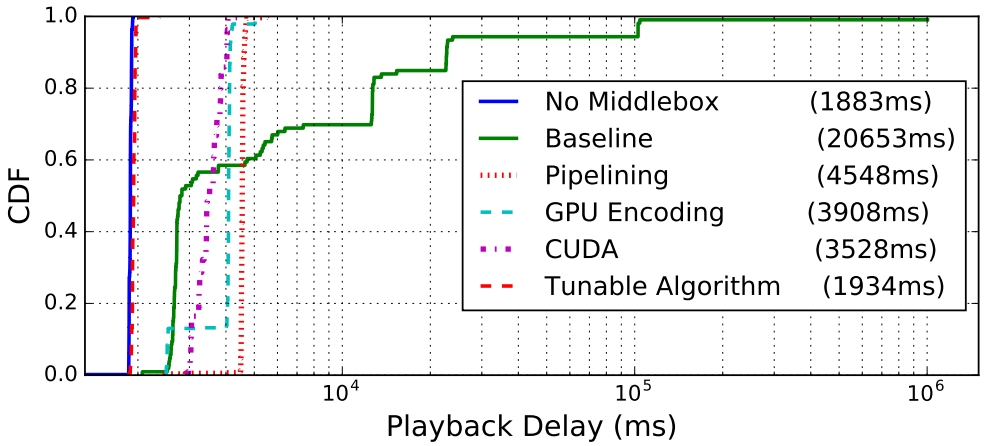

Figure 6.2: CDF of Playback Delay

6.4 Element Processing Time

To understand how the workload is distributed among pipeline stages, we measured processing time for each GStreamer element by parsing the `GST_SCHEDULING` log. Similar to measurements on the playback delay, we collected data for each step of the optimization. Since the baseline system cannot handle 1080p video at 25 FPS, we reconstructed the video at 4 FPS to prevent frame loss during evaluation. Moreover, since data blocks processed by each element have different units, such as UDP packet for `Transport`, H.264 packet for `Decode` and frame for `Processing`, we aggregated processing times for the entire video and divided it by the total number of frames to show element processing time per frame.

As shown in Figure 6.3, each optimization step mostly reduces the element processing time, with the only exception for CUDA. Although enabling CUDA reduces the overall playback delay, it increases the time for face detection. This could be due to the memory copy overhead between CPU RAM and GPU VRAM. However, since outsourcing the Processing stage to CUDA frees CPU resources for other stages, the overall system enjoys a reduction in playback delay. Lastly, we also observe that GPU encoding significantly reduces the time

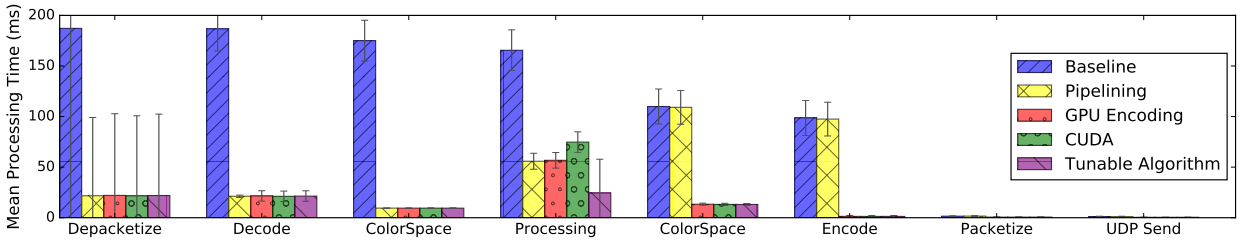

Figure 6.3: Element Processing Time Per Frame

taken by not only Encoding, but also the ColorSpace conversion stage before it. This is because our GPU encoder accepts the same color format as the Processing stage output, eliminating the need to perform a color space conversion.

6.5 Functional Evaluation with Tuning

To evaluate how varying Skip Frame (T) and CAMShift frame (M) changes MiBao's behavior, we fed MiBao with the same video multiple times at different T and M values, and measured various metrics of the system. The video is played at 8 FPS to leave enough time (125ms) for processing each frame, more than enough even at T=0 and M=0.

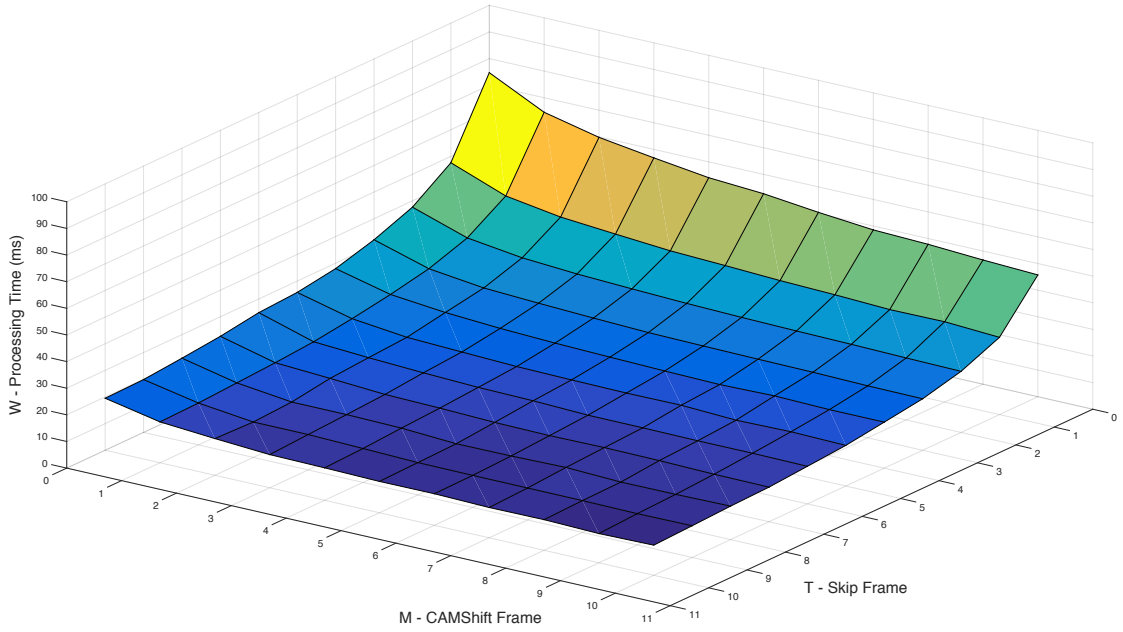

Figure 6.4: Processing Time W vs T and M

Mean Processing Time vs T and M

Figure 6.4 confirms our expectation that Mean Processing Time is inversely related to T and M. Moreover, the result also shows T has a greater impact on the mean processing time

than that of M. This is expected because skipping a frame is still less time consuming than doing CAMShift.

Quantify Accuracy

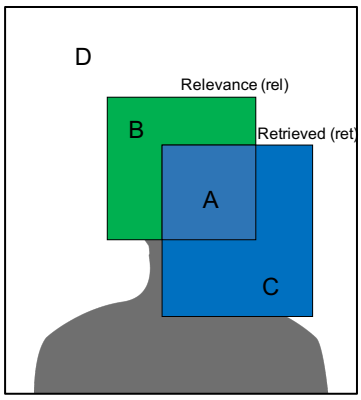

Figure 6.5: Face Detection Bonding Boxes. A is True Positive (TP); C is False Positive (FP); B is False Negative (FN); D is True Negative (TN)

The results of a face detection algorithm is given as a list of bounding boxes enclosing the detected areas of faces. As shown in Figure 6.5, the Retrieved (ret) box represents MiBao's detection result, whereas the Relevance (rel) box represents the reference detection result. We obtained the reference results using the Google Cloud Vision API because we assume that Google's API should produce much more accurate detection results than ours. Both MiBao's and the reference's detection results are logged frame-by-frame to a local log, indicating the frame index, number of faces, and list of bounding boxes. We then compared both log to obtain an accuracy value for each frame, and average them for the entire video to obtain the mean accuracy.

As discussed in Section 3.1, an accurate face detection algorithm should have very few false negatives and false positives. To quantify accuracy based on this requirement, we first adopted the **Accuracy** formula [47]:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Using labels in Figure 6.5 and the relationship between rel and ret boxes:

$$\text{Accuracy} = \frac{A + D}{A + B + C + D} = \frac{\text{rel} \cap \text{ret} + (\text{Total Area} - \text{rel} \cup \text{ret})}{\text{Total Area}}$$

However, although this formula can give a normalized detection accuracy for a frame, it does not serve our evaluation well. As the accuracy formula has true negative area D in both numerator and denominator, when faces only cover a small portion of the frame, D

dominates the formula and forces accuracy values close to 100%. This prohibits us from examining the variations in accuracy when changing T and M. Instead, a better metric for accuracy in our context is **F-Measure** [55] that is a combination of other two metrics, **Recall** and **Precision**. Recall measures how much relevant detection is captured by our detection and is defined as:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{A}{A + B} = \frac{\text{ref} \cap \text{ret}}{\text{rel}}$$

Precision measures how much of our detection is part of the relevance detection, and is defined as:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{A}{A + C} = \frac{\text{ref} \cap \text{ret}}{\text{ret}}$$

F-Measure is the harmonic mean of the two metrics so that we take both over-detection and under-detection into account. It is defined as:

$$F = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Therefore, in our experiments, "accuracy" is defined as **the F-measure of our face detection algorithm compared to Google's Cloud Vision API**.

F-Measure vs T and M

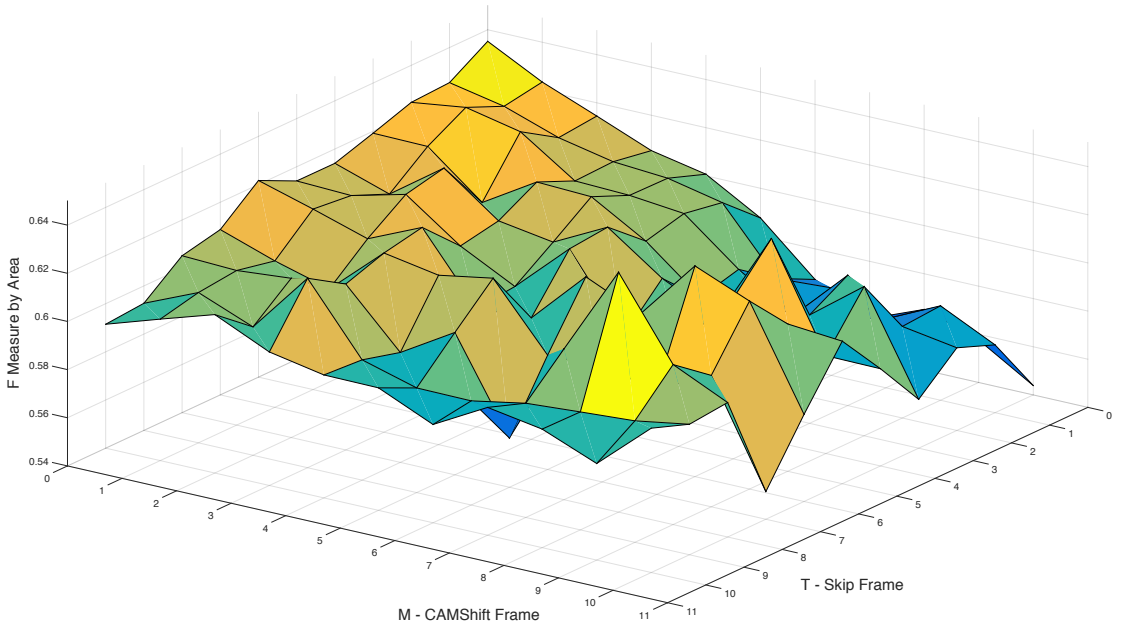

Figure 6.6: F-Measure F vs T and M

Figure 6.6 shows that F-measure varies significantly with T and M, and has general inverse relations with both parameters. In particular, F-Measure varies more with respect

to changes in CAMShift Frames (M) than with respect to changes in Skip Frames (T). This may likely be caused by how CAMShift fails to track a face by color without frequent-enough calibration through Viola-Jones face detection, and faces do not appear and disappear in the video often enough to penalize Frame Skip. Combining the findings from Section 6.5, we suggest that when tuning the algorithm to save processing time, Skip Frame (T) should be tuned in favor of CAMShift Frame (M).

F-Measure vs Mean Processing Time

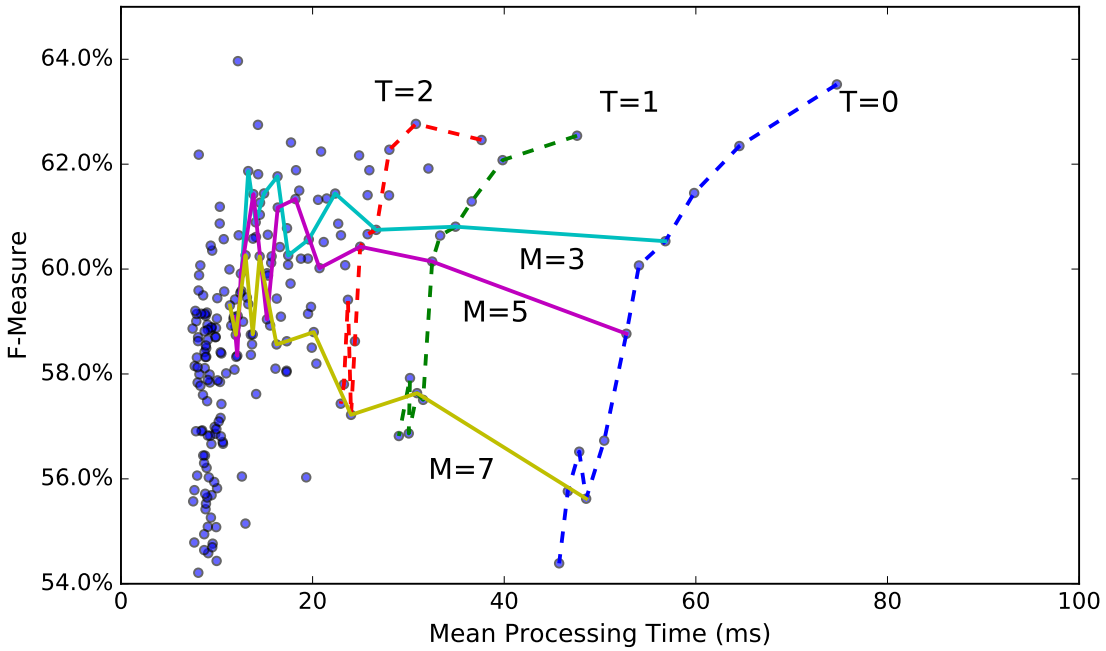

Figure 6.7: F-Measure F vs Processing Time W

Figure 6.7 shows a general direct relation between the F-measure and mean processing time. There is a saturation region towards the right tail as our algorithm approaches its accuracy limit. Moreover, at constant T value (dashed lines), increasing M without T results in a more than proportional drop in F-measure than savings in mean processing time. On the other hand, when increasing T at constant M (solid lines), we enjoy a more than proportional saving in mean processing time than the drop in F-measure. This confirms our previous conclusion that Skip Frame (T) should be tuned in favor of CAMShift frame (M).

6.6 Scalability Evaluation with Tuning

To test how MiBao behaves at scale, we conduct experiments by starting multiple Video Processor instances at the same time. For each Video Processor instance, we feed it with 30s

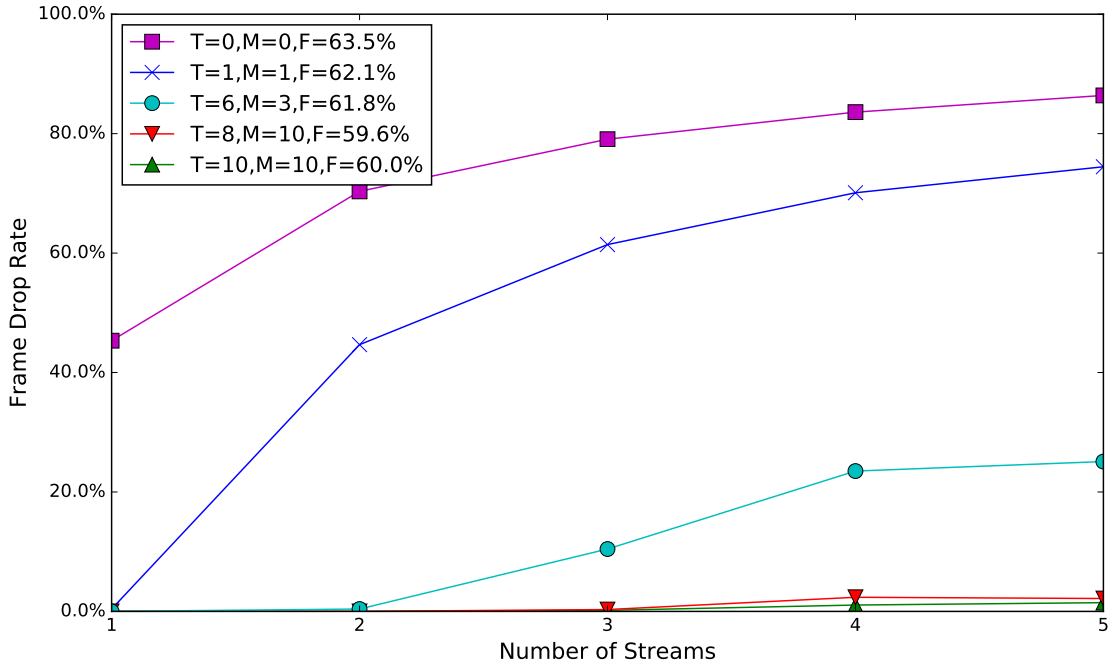

Figure 6.8: Frame Drop Rate vs Number of Streams

of test videos at 25 FPS. We start each video at 1 second succession so that the contents of videos are not in-phase with each other in order to simulate real life use case. At 5 Mbit/s per video stream and 1 Gbps network bandwidth, our test range, 1 to 10 streams, does not make network bandwidth a limiting factor.

As shown in Figure 6.8, at T = 10, M = 10, MiBao can support 4 streams at 1% frame loss and 60.0% accuracy by F-measure, with Precision = 76.8% and Recall = 65.0%. With fewer streams, MiBao can be tuned to be more accurate. For example, with two streams, MiBao can run at T=6 and M=3 with less than 1% frame loss and F-measure of 61.8%.

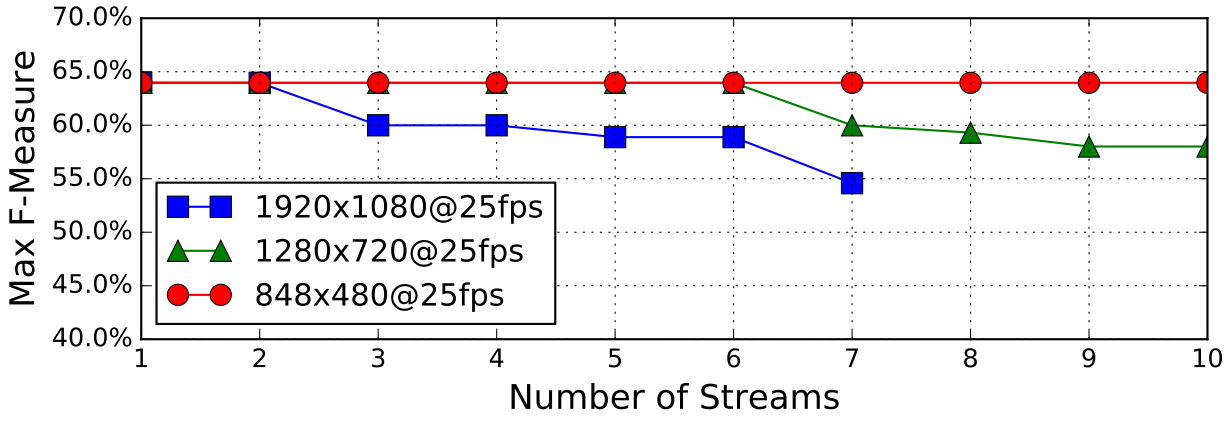

Figure 6.9: Accuracy vs Number of Stream

Figure 6.9 illustrates the maximum accuracy by F-measure MiBao can provide with less than 1% frame drop, given the increasing number of streams. Under smaller resolutions such as 720p and 480p, MiBao is able to support more streams with more than 60% accuracy by F-measure. Because of the 4 GB VRAM limitation, we could test at most seven 1080p streams simultaneously on a g2.2xlarge instance.

Multiple GPUs

Since our middlebox relies heavily on GPU for its computation, we conducted a scalability evaluation on a g2.8xlarge instance that has 4 times GPU, CPU and RAM resources than the g2.2xlarge instance. It has a 10 GbE network link, making network bandwidth less of a problem even with twenty 5 Mbit/s video streams. As shown in Figure 6.10, with 4x resources, MiBao can process 4 streams at T=1, M=1 with F=62.1%, 8 streams at T=6, M=3 with F=61.8%, and 12 streams at T=8, M=10 with F=59.6%.

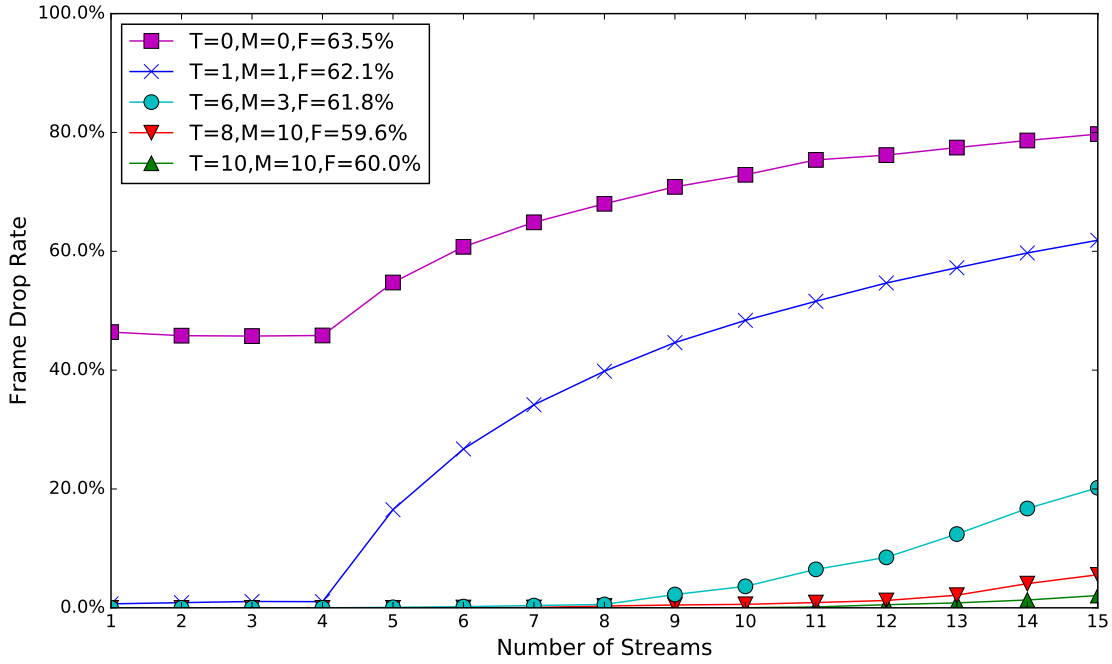

Figure 6.10: Frame Drop Rate vs Number of Streams

Chapter 7

Future Work

7.1 Online Algorithm Tuning

Currently, we determine algorithm parameters using profiling results from the past. While this provides a quick way of generating load based parameters, the existing profile may not represent the behavior of future workloads. Ideally, a control-loop should be developed to actively change the two parameters with respect to the input queue size. When the input queues of the Video Processor instances get larger, the Middlebox Daemon or the Video Processor itself should increase T or M to speed up the processing rate and maintain the queue at a reasonable size.

7.2 NIC-GPU Memory Optimization

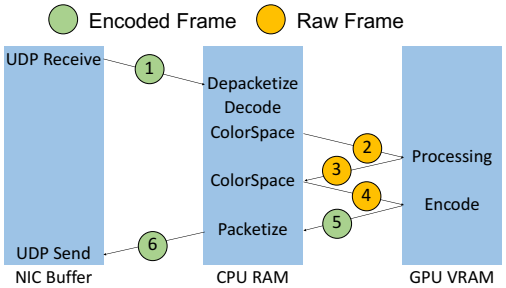

Figure 7.1: Current Memory Copy Among Devices

As our middlebox uses NIC, CPU and GPU at different stages, data is copied back and forth among NIC Buffer, CPU RAM, and GPU VRAM. As shown in Figure 7.1, a video packet is copied 6 times among devices in the Video Processor. Among them, three copies are for encoded video frames and three copies are for raw video frames. While copying encoded streams takes small amount of time, copying raw frames can take much longer. For example,

as the PCI-E 3.0 x16 bus used by the NVIDIA Grid K520 graphic card has a bandwidth of 16 GB/s [16], a 6.22 MB 1080p raw frame needs at least 0.38 ms to be copied through that bus and copying it three times adds at least 1 ms to the playback delay. The actual overhead can be much larger as 16 GB/s is only the theoretical maximum and does not take into account the start-up and tear-down cost of an inter-device data transfer operation.

To reduce such overhead, we could first implement stages between `Decode` and `Encode` using GPU based solutions [37, 48] so that we only copy encoded frames to and from the GPU, as shown in Figure 7.2. We could then implement algorithms to depacketize and packetize RTP streams on the GPU. When all stages are implemented on the GPU, we could take advantage of existing NIC-GPU memory optimizations to directly copy data between the NIC buffer and VRAM [32, 66], reducing the number of memory copies to two as shown in Figure 7.3. Given the playback delay of our current system, such an optimization may enable Video Processing on Type 3 applications.

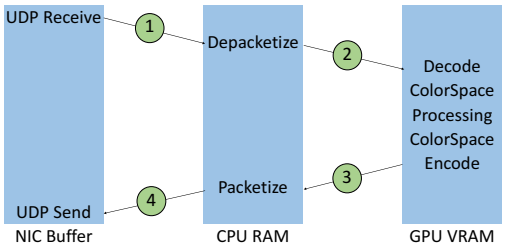

Figure 7.2: Memory Copy with GPU Based Video Processing

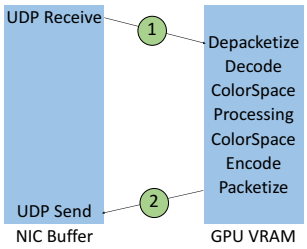

Figure 7.3: NIC-GPU Memory Optimization

Chapter 8

Related Work

Our work contributes to and was inspired by a multidisciplinary group of researches in middleboxes, Software-Defined Networking (SDN), computer vision algorithms, hardware accelerated network devices.

8.1 Middlebox

The core of our work touches multiple aspects of middlebox technologies including packet filters, deep packet inspection, and media gateways.

Packet Filters: Our middlebox relies on packet filters to extract video streams from network traffic using the address and port number 4-tuple. In the past, a variety of user space and kernel level packet filters were invented [44, 19, 22], allowing us to create packet filters on commodity machines with ease. Recently, the development of SDN based approach such as OpenFlow [45] makes our deployment even easier. Currently, we use the Click Software Router [39] because it is easy to set up and our main focus is on the Video Processor instead of the Packet Filter. In the future, the Packet Filter could be implemented more efficiently in kernel space or on hardware.

Deep Packet Inspection: The motivations of our work lie in parallel with the development of Deep Packet Inspection (DPI). Researchers have invented Intrusion Detection Systems (IDS) [2, 56] to protect local users from exposing to inappropriate or dangerous content. There are also Exfiltration prevention systems [12, 43, 61] that protect a local network from accidental loss of private data. However, most of these applications are limited to fixed field processing and textual pattern matching, and are not capable of performing complex operations on visual semantic content in video streams.

Media Gateways: Researchers invented media gateways that are able to transcode voice and video formats [59, 34, 17]. This kind of middlebox is useful in reducing end-host workload for IPTV and video conferencing applications. However, a media gateway does not analyze media content. Even if it does, it does not use as equally sophisticated computer

vision algorithms as described in this paper and therefore is not functionally equivalent to our middlebox.

8.2 Software-Defined Networking

While our work introduces a new network application, it is not useful unless deployed. The rise of Software-Defined Networking (SDN) technologies eases the deployment of our application. Software enabled network switches such as OpenFlow [45] and BESS [31] allow packet filters to be fully programmable. With this, the Gateway Deamon can directly modify the flow table to redirect video traffic to its Video Processor without going through the Gateway Daemon. Moreover, recent developments in Network Function Virtualization (NFV) [49, 27] help facilitate the deployment of new network applications by handling low level resources management. However, we observe that most NFV frameworks do not allocate GPU resources. With more network appliances that use GPU, there is a greater need for GPU provisioning through NFV platforms.

8.3 Computer Vision Algorithms

Our work replies on recent developments in computer visions algorithms such as the Viola-Jones Face Detection Algorithm [68, 67] and the OpenCV library [10]. However, to our knowledge, very few computer visions algorithms were specially designed for network appliances. Thus, they did not take into account the requirement on processing time and scalability. Our work demonstrates that, a tunable computer vision algorithm can maximize resource usage in a middlebox while meeting the processing time constraint.

8.4 Hardware Accelerated Network Devices

Past research projects like PacketShader [32] have already integrated the GPU into network appliances. They exploit GPU's parallelism to accelerate traditional network operations. However, few of these projects explore the aspect of applying computer vision algorithms on network traffic. While middleboxes in these projects do not provide the same function as ours, we could utilize their memory optimization techniques to speed up our computation, as described in Section 7.2.

Chapter 9

Conclusion

We have presented MiBao, a Video Processing Middlebox that performs face detection and blur on an H.264 encoded RTP-UDP video stream. Pipelining, GPU acceleration, and a tunable context-aware face detection algorithm significantly reduce the processing time and improve the throughput of our middlebox. Using an AWS g2.2xlarge instance, our middlebox is able to process up to four 1080p video streams at 25 FPS with less than 100 ms playback delay and more than 60% accuracy by F-measure.

As a proof of concept, our middlebox shows that video stream processing on the wire using computer vision algorithms is not only feasible but also scalable. Deployment of such middlebox will alleviate the concern over privacy leakage and inappropriate content exposure through video streaming applications.

Bibliography

[1] *Amazon EC2 T2 Instances*. https://aws.amazon.com/ec2/instance-types/t2/.

[2] *Bro*. http://www.bro.org.

[3] *FFMpeg*. http://www.ffmpeg.org.

[4] *GLib*. https://developer.gnome.org/glib/.

[5] *How your camera works with Home/Away Assist*. https://nest.com/support/article/How-your-camera-works-with-Home-Away-Assist.

[6] *HTTP Dynamic Streaming*. http://www.adobe.com/products/hds-dynamic-streaming.html.

[7] *Microsoft Silverlight*. https://www.microsoft.com/silverlight/.

[8] *Nest Cam*. https://nest.com/camera/meet-nest-cam/.

[9] *NVIDIA VIDEO CODEC SDK*. https://developer.nvidia.com/nvidia-video-codec-sdk.

[10] *OpenCV*. http://opencv.org/.

[11] Rahul Patel. *Real Time Face Detection Using Viola Jones and CAMShift in Python*. https://rahullpatell.wordpress.com/2015/04/21/real-time-face-detection-using-viola-jones-and-camshift-in-python-i/. Blog. 2015.

[12] *Symantec Data Loss Prevention*. https://www.symantec.com/products/information-protection/data-loss-prevention.

[13] *VLC*. http://www.videolan.org/.

[14] *x264 Codec*. http://www.videolan.org/developers/x264.html.

[15] *YouTube*. http://www.youtube.com.

[16] Jasmin Ajanovic. "PCI express 3.0 overview". In: *Proceedings of Hot Chip: A Symposium on High Performance Chips*. 2009.

[17] Flemming Andreasen and Bill Foster. *Media gateway control protocol (MGCP) version 1.0*. Tech. rep. 2002.

[18] Gary Audin. "Next-Gen Firewalls: What to Expect". In: *BUSINESS COMMUNICATIONS REVIEW*. 34.6 (2004), pp. 56–61.

[19] Mary L Bailey, Burra Gopal, Michael A Pagels, Larry L Peterson, and Prasenjit Sarkar. "PathFinder: A Pattern-Based Packet Classifier." In: *OSDI*. 1994, pp. 115–123.

[20] Mario Baldi and Yoram Ofek. "End-to-end delay analysis of videoconferencing over packet-switched networks". In: *Networking, IEEE/ACM Transactions on* 8.4 (2000), pp. 479–492.

[21] Salman A Baset and Henning Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol". In: *arXiv preprint cs/0412017* (2004).

[22] Andrew Begel, Steven McCanne, and Susan L Graham. "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture". In: *ACM SIGCOMM Computer Communication Review*. Vol. 29. 4. ACM. 1999, pp. 123–134.

[23] David Belson. *Akamai's State of the Internet Q4 2015 Report*. report. Akamai Technologies, Inc., Mar. 2016.

[24] Omer Boyaci, Andrea Forte, Salman Abdul Baset, and Henning Schulzrinne. "vDelay: A tool to measure capture-to-display latency and frame rate". In: *Multimedia, 2009. ISM'09. 11th IEEE International Symposium on*. IEEE. 2009, pp. 194–200.

[25] Gary R Bradski. "Real time face and object tracking as a component of a perceptual user interface". In: *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*. IEEE. 1998, pp. 214–219.

[26] Kris Brosch. *Reversing the Dropcam Part 2: Rooting your Dropcam*. `http://blog.includesecurity.com/2014/04/reverse-engineering-dropcam-rooting-the-device.html`. Apr. 2014.

[27] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. "Ethane: taking control of the enterprise". In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. 4. ACM. 2007, pp. 1–12.

[28] Federal Trade Commission et al. "Internet of Things: Privacy & security in a connected world". In: *Washington, DC: Federal Trade Commission* (2015).

[29] Somak R Das, Silvia Chita, Nina Peterson, Behrooz A Shirazi, and Medha Bhadkamkar. "Home automation and security for mobile devices". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*. IEEE. 2011, pp. 141–146.

[30] Becky Freeman and Simon Chapman. "Is "YouTube" telling or selling you something? Tobacco content on the YouTube video-sharing website". In: *Tobacco Control* 16.3 (2007), pp. 207–210.

[31] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. EECS Department, University of California, Berkeley, May 2015. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html`.

[32] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. "PacketShader: a GPU-accelerated software router". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 195–206.

[33] Rhys Hill, Christopher Madden, Anton van den Hengel, Henry Detmold, and Anthony Dick. "Measuring latency for video surveillance systems". In: *Digital Image Computing: Techniques and Applications, 2009. DICTA'09.* IEEE. 2009, pp. 89–95.

[34] Hao Hou, Yong He, Tuan Minh Nguyen, and Ahmed Doleh. *Methods and systems for providing lawful intercept of a media stream in a media gateway.* US Patent 7,092,493. Aug. 2006.

[35] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. "Analyzing forged ssl certificates in the wild". In: *Security and privacy (sp), 2014 ieee symposium on.* IEEE. 2014, pp. 83–97.

[36] Open Net Initiative et al. *YouTube censored: A recent history.* 2011.

[37] Frank Jargstorff and Eric Young. "CUDA Video Decoder API". In: *NVIDIA Corporation* 2 (2008).

[38] Jeff Jarmoc and Dell SecureWorks Counter Threat Unit. "SSL/TLS interception proxies and transitive trust". In: *Black Hat Europe* (2012).

[39] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[40] Wouter J Kooij, Hans M Stokking, Ray van Brandenburg, and Pieter-Tjerk de Boer. "Playout delay of TV signals: measurement system design, validation and results". In: *Proceedings of the 2014 ACM international conference on Interactive experiences for TV and online video.* ACM. 2014, pp. 23–30.

[41] Pui Y Lee, Siu C Hui, and Alvis Cheuk M Fong. "Neural networks for web content filtering". In: *Intelligent Systems, IEEE* 17.5 (2002), pp. 48–57.

[42] Fuwen Liu and Hartmut Koenig. "A survey of video encryption algorithms". In: *computers & security* 29.1 (2010), pp. 3–15.

[43] Simon Liu and Rick Kuhn. "Data loss prevention". In: *IT professional* 12.2 (2010), pp. 10–13.

[44] Steven McCanne and Van Jacobson. "The BSD packet filter: A new architecture for user-level packet capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings.* USENIX Association. 1993, pp. 2–2.

[45] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[46] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. "MOT16: A Benchmark for Multi-Object Tracking". In: *arXiv:1603.00831 [cs]* (Mar. 2016). arXiv: 1603.00831. URL: http://arxiv.org/abs/1603.00831.

[47] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[48] David Oro, Carles Fernández, Javier Rodríguez Saeta, Xavier Martorell, and Javier Hernando. "Real-time GPU-based face detection in HD video sequences". In: *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*. IEEE. 2011, pp. 530–537.

[49] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. "E2: a framework for NFV applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 121–136.

[50] Roger Pantos and William May. *HTTP Live Streaming*. Internet-Draft draft-pantos-http-live-streaming-19. Work in Progress. Internet Engineering Task Force, Apr. 4, 2016. 54 pp. URL: https://tools.ietf.org/html/draft-pantos-http-live-streaming-19.

[51] H Parmar and M Thornburgh. "Adobe's Real Time Messaging Protocol". In: *Copyright Adobe Systems Incorporated* (2012), pp. 1–52.

[52] Jason Robert Carey Patterson. "Video Encoding Settings for H.264 Excellence". In: (Apr. 2012).

[53] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. "Real-time computer vision with OpenCV". In: *Communications of the ACM* 55.6 (2012), pp. 61–69.

[54] Kristin Purcell. *The state of online video*. Pew Internet & American Life Project Washington, DC, 2010.

[55] C. J. Van Rijsbergen. *Information Retrieval*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979. ISBN: 0408709294.

[56] Martin Roesch et al. "Snort: Lightweight Intrusion Detection for Networks." In: *LISA*. Vol. 99. 1. 1999, pp. 229–238.

[57] H. Schulzrinne and S. Casner. *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 3551. RFC Editor, July 2003. URL: http://www.rfc-editor.org/info/rfc3551.

[58] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. Oct. 14, 2015. DOI: 10.17487/rfc3550. URL: https://rfc-editor.org/rfc/rfc3550.txt.

[59] Vyas Sekar, Sylvia Ratnasamy, Michael K Reiter, Norbert Egi, and Guangyu Shi. "The middlebox manifesto: enabling innovation in middlebox deployment". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, p. 21.

[60] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making middleboxes someone else's problem: network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[61] George J Silowash, Todd Lewellen, Daniel L Costa, and Todd B Lewellen. "Detecting and preventing data exfiltration through encrypted web sessions via traffic inspection". In: (2013).

[62] Ray Smith. "An overview of the Tesseract OCR engine". In: *icdar*. IEEE. 2007, pp. 629–633.

[63] GStreamer Team. *GStreamer: open source multimedia framework*.

[64] Saurabh Tewari and Leonard Kleinrock. "Analytical model for BitTorrent-based live video streaming". In: *Proc. IEEE NIME Workshop*. 2007.

[65] Gareth Tyson, Yehia Elkhatib, Nishanth Sastry, and Steve Uhlig. "Demystifying porn 2.0: A look into a major adult video streaming website". In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 417–426.

[66] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. "GASPP: a GPU-accelerated stateful packet processing framework". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 321–332.

[67] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2001, pp. I–511.

[68] Paul Viola and Michael J Jones. "Robust real-time face detection". In: *International journal of computer vision* 57.2 (2004), pp. 137–154.

[69] Y-K Wang, R Even, T Kristensen, and R Jesup. *RTP payload format for H. 264 video*. Tech. rep. 2011.

[70] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya-Qin Zhang, and Jon M Peha. "Streaming video over the Internet: approaches and directions". In: *Circuits and Systems for Video Technology, IEEE Transactions on* 11.3 (2001), pp. 282–300.