ePrivateEye: To the Edge and Beyond!

Christopher Streiffer, Animesh Srivastava, Victor Orlikowski, Yesenia Velasco,
Vincentius Martin, Nisarg Raval, Ashwin Machanavajjhala, Landon P. Cox
Computer Science, Duke University, USA

ABSTRACT

Edge computing offers resource-constrained devices low-latency access to high-performance computing infrastructure. In this paper, we present ePrivateEye, an implementation of PrivateEye that offloads computationally expensive computer-vision processing to an edge server. The original PrivateEye locally processed video frames on a mobile device and delivered approximately 20 fps, whereas ePrivateEye transfers frames to a remote server for processing. We present experimental results that utilize our campus *Software-Defined Networking* infrastructure to characterize how network-path latency, packet loss, and geographic distance impact offloading to the edge in ePrivateEye. We show that offloading video-frame analysis to an edge server at a metro-scale distance allows ePrivateEye to analyze more frames than PrivateEye's local processing over the same period to achieve realtime performance of 30 fps, with perfect precision and negligible impact on energy efficiency.

1 INTRODUCTION

Image processing is increasingly important for mobile applications, such as augmented-reality, mobile-gaming, and video-streaming. Modern system-on-a-chip (SoC) designs have undergone rapid improvements in CPU and GPU processing capabilities, but, as a result of power and space constraints, these platforms cannot duplicate the parallelism, straight-line speed, or large memories of high-end servers. For example, a high-end GPU may consume 500 Watts of power, whereas a high-end SoC GPU will consume fewer than 10 Watts. Similarly, high-end smartphones rarely offer more than 4GB of RAM, whereas mid-range servers offer 128GB of RAM. This resource gap significantly limits the quality of mobile image-processing applications.

SEC '17, San Jose / Silicon Valley, CA, USA
© 2017 ACM. 978-1-4503-5087-7/17/10…$15.00
DOI: 10.1145/3132211.3134457

Edge computing is an approach to compute infrastructure that places small data centers geographically close to clients. In recent years, both industry and academia have undertaken initiatives related to edge computing, including cloudlets [19], fog computing [3], and micro data centers [1]. The goal of edge computing is to give resource-constrained clients access to greater compute, memory, and storage resources than are available locally, and to provide that access over a higher-quality connection than is possible over the wide area. Prior research has sought to close the performance and energy gaps between resource-poor devices and server-side infrastructure through remote execution and code offload [2, 6–8, 13, 15, 16], and image-processing applications provide natural use cases for offloading computation to edge servers.

Edge computing promises to address one of the biggest problems with code offload for realtime image processing: network latency. Realtime image-processing applications often require results within tens of milliseconds, and many campus, municipal, and regional networks offer both the capacity (e.g., fiber-optic networks) and software-defined networking (SDN) orchestration to establish low-latency and high-bandwidth paths between sensors and edge clusters.

The PrivateEye [18] access-control framework for visual information is an example of a system that could benefit from edge computing. PrivateEye defines a generic two-dimensional special shape that is easy for users to draw, e.g., on a piece of paper, on a whiteboard, or within a projected presentation, and easy for software on a recording device to identify in realtime. The original PrivateEye implementation performed all marker-recognition locally on a mobile device and could process frames at a rate of 20 fps.

In this paper, we present an edge-enabled version of PrivateEye called *ePrivateEye*. ePrivateEye differs from the original PrivateEye in one significant facet: it offloads marker detection to an edge server. We explore different network and server configurations for running ePrivateEye and provide benchmark evaluations for each. In specific, we measure frames per second (fps), battery consumption, and precision/recall under ePrivateEye. The three configurations consist of marker detection running locally on the mobile device, running on a wireless access point, and running on a GPU server connected directly to the access point. We also perform experiments with our campus SDN to control network behavior in order to understand how "well provisioned" the path between a recording device and an edge server must be to support ePrivateEye.

We envision the system running within an enterprise setting where SDN could be utilized to provide secure, fast-path access to the edge server running ePrivateEye.

Our contributions can be summarized as follows:

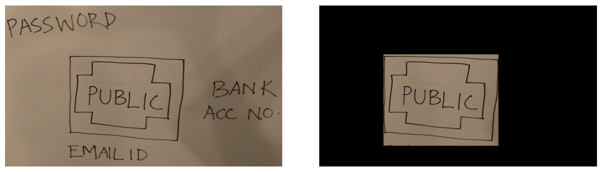
- We evaluate ePrivateEye under several network and server configurations, including offloading work to an access point, to an in-building server, and to an off-campus server.
- We leverage our campus SDN to evaluate ePrivateEye under metro-scale geographic distances between a device and server.
- We show that offloading marker-detection to an edge server at a metro-scale distance allows ePrivateEye to analyze 50% more frames than PrivateEye's local processing over the same period, with perfect precision and negligible impact on energy efficiency.

The remainder of the paper is structured as follows. Section 2 provides background on PrivateEye and the motivation behind running the service on an edge server. We provide the inner workings of PrivateEye in Section 3. Section 4 presents the design principles that guided our work. Section 5 provides system implementation details, and discusses how SDN can be utilized to improve network performance. Sections 6, 7 and 8 provide the results from our experiments, explores the feasibility of ePrivateEye on SDN and cloud infrastructure, respectively. Next, Section 9 presents a brief discussion on ePrivateEye in the context of SDNs and cloud. Section 10 provides insight on related work, and details other applications that have shown success at offloading processing functionality. Finally, Section 11 provides the conclusion of our work.

2 BACKGROUND AND MOTIVATION

With the ever-more evident need for sensitive data protection in businesses and the provision of privacy guarantees to application users, PrivateEye has shown to be a viable solution in protecting data in real time. PrivateEye uses advanced computer vision methods to detect and block regions within a frame that contain sensitive information. Integrated within the Android camera subsystem, PrivateEye functions as middleware between the camera driver and applications accessing the camera service. PrivateEye is able to deliver frames at a median rate of 20 FPS [18], providing reasonable quality of service for users while simulataneously blocking sensitive regions within the video frames.

A deeper look inside PrivateEye's system design reveals that PrivateEye runs in two modes: (1) Detection, and (2) Tracking. In detection mode, PrivateEye performs analysis on incoming frames to determine which regions have been marked public by the user. During tracking mode, PrivateEye finds the location of the last detected public region in incoming frames. While detection mode is computationally



(a) (b)

Figure 1: (a) A region on a whiteboard marked as public using PrivateEye's marker. (b) An app's view while capturing the same region using PrivateEye's software enabled smartphone.

intensive (slow), tracking is conversely light on computation (fast). As a result, PrivateEye runs in detection mode periodically (on 10% of the frames) and employs tracking mode in the intervening intervals.

While this method of operation helps PrivateEye to achieve a high frame delivery rate, the use of tracking inherently results in occasional inaccuracies in determining the location of the public region and causes a small drop in recall. Furthermore, since computer vision algorithms are imperfect, the use of tracking can result prolonged exposure of a region that was wrongly judged as public during the most recent execution of detection mode.

Edge computing [10] offers an alternative to centralized computing, in that resource-constrained devices are able to connect to powerful, geographically-proximal endpoints. Scenarios like home and enterprise networks can be tuned to better serve clients. Such an edge-computing based solution can significantly overcome the issues faced by PrivateEye. Specifically, the compute intensive operations can be performed remotely enabling continuous detection mode execution. Also, a good connectivity between the edge server and the client can ensure higher rate of frame delivery. This forms the motivation behind exploring the feasibility of PrivateEye as an edge computing based system, ePrivateEye.

3 PRIVATEEYE OVERVIEW

Before we dive into the details of ePrivateEye's system design, we provide some details of PrivateEye's inner workings. PrivateEye is a system whose aim is to provide more control over the information being accessed by a third-party app to the user. PrivateEye focuses on two-dimensional regions and takes a privacy marker approach [17] wherein a user marks a two-dimensional area as a public region using a special marker as shown in Figure 1a. Subsequently, when a camera-enabled application captures the same two-dimensional region when using a PrivateEye enabled smart device, PrivateEye intercepts the image data obtained by the camera, detects the marked region, blocks the remainder of the region, and delivers the data to the application (Figure 1b).

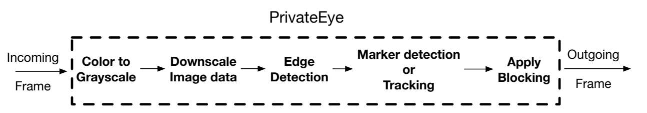


Figure 2: Series of computer vision algorithms applied on image data under PrivateEye.

Inside PrivateEye's logic for discerning public and private regions, each individual camera frame undergoes significant processing. First, the colorspace of the frame is converted from color to grayscale; most computer vision algorithms operate in grayscale in order to achieve faster results. Next, the grayscale image is scaled down to a size of 800×480 for further processing. PrivateEye detects edges in the frame using the well-known Canny [5] algorithm. After edge detection, the system detects contours in the edge-based image using Suzuki's algorithm [22]. The algorithm returns the contours in a tree data structure which preserves the contour hierarchy. PrivateEye runs an approximation to find a parent contour that can be approximated as a rectangle and a child contour that has 12 corners with area at least 50% of the parent rectangle contour. This is the region marked by the user. Finally, the system finds the bounding box for the marked region and blocks everything else in the image.

As mentioned earlier, since the marker detection algorithm is slow, PrivateEye runs the detection algorithm periodically. For other frames, the system employs tracking on the last observed marked region in the current frame using the Lucas-Kanade optical-flow-in-pyramids technique [4]. Figure 2 depicts the series of transformations a frame experiences under PrivateEye.

The entire PrivateEye module is integrated into the trusted portion of Android itself; PrivateEye is contained within the camera service, which resides between the camera sensor and the application framework. Every frame captured from the camera sensor is received by the camera service, before being sent to the associated application. PrivateEye monitors the frames received by camera service and applies all required image transformations before sending the frame to the application.

ePrivateEye is an extension of PrivateEye and uses the same techniques employed in PrivateEye. To get more details about the PrivateEye, we would encourage the readers to read the original work [18].

4 DESIGN PRINCIPLES

In this section we present the design principles behind ePrivateEye. Our primary goal was to achieve near real-time performance for camera apps. With this as our motivation, we designed the server and client component of ePrivateEye to minimize network latency, while also reducing traffic size.

Reduce Client-side Computation: Since the client runs on a resource constrained device, we identified all the compute-intensive functionality and offloaded it. When a frame is received by the ePrivateEye client running inside the camera service, it is processed in the following fashion: (1) A copy of the frame data is saved, and a numeric identifier associated to it, (2) A grayscale color conversion is performed, and (3) The grayscale image data is downsized to a resolution of 400×240. The remainder of the image transformations shown in Figure 2 are part of the server component of ePrivateEye. After subsequent processing by the server that results in the marked region information being available for the frame, the client blocks the region outside the marked region and returns the modified frame data to the camera service.

Reduce Network Traffic: Again, since the ePrivateEye client runs on a resource constrained device, it is often the case that the software on such devices is designed to save power by reducing the amount of data transmitted, where possible. In our case, sending a frame to the ePrivateEye server at its original resolution (e.g. 1280×960) would cost significant time and energy, thereby negating some of the gains we would hope to achieve through offloading processing. We instead chose to reduce the amount of traffic sent by the client to the server by performing the sequence of relatively compute-inexpensive image transformations described above before sending the frame to the server. Our original frame data was encoded in NV12 format frames. Since edge detection performs satisfactorily on grayscale images, each frame is downsampled to this format, resulting in a size savings of 33%. We then further downsample the data by converting it to a resolution of 400×240, as the computer vision algorithms we use provide acceptable performance at that resolution. Through these transformations, we reduce the size of the data transferred by the client from $1280 \times 960 \times 1.5$ to 400×240.

On the server side, we reduce the amount of traffic sent to the client by returning the coordinates of the detected regions only, rather than the masked frames.

By making the design choice to perform image blocking transformations on the client, we were able to reduce the total amount of traffic significantly. As mentioned earlier, these operations have a small computational overhead; the reduction in transferred data size makes the slightly increased computational cost of retaining these operations on the client a worthwhile trade.

Minimize Latency: PrivateEye is implemented to interact in a non-blocking fashion with the camera service. In this design, the camera service submits a frame to PrivateEye for processing, before making a request for the most recently processed frame. If the frame is available, PrivateEye will return the masked frame to the service. Otherwise, no frame is returned and the camera service will wait for the next frame to be received from the camera sensor.

To minimize latency between the client and server, we made several modifications to the TCP sockets maintained by each. On the client side, we disable Nagle's algorithm on each socket. Nagle's algorithm purposefully delays frames in order to reduce the number of TCP packets sent over the network, and thereby adversely affects latency sensitive applications. Disabling the algorithm forces packets to be sent over the socket as soon as they are received, thus reducing latency. On the server side, we look to reduce the number of `read` syscalls made by the server by modifying the receive buffer. We only allow for the server to make a `read` on a socket once a certain number of bytes have arrived. We set the number of bytes to be equivalent to the total size of the expected frame.

To minimize latency when transmitting across the metropolitan network, we utilize Software Defined Networking (SDN) to control the path along which the data is transmitted. This allows for a "fast path" to be established between the access point (AP) and server. This implementation will be discussed in more detail in the following section.

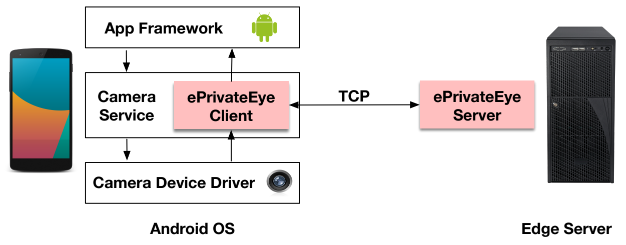


Figure 3: ePrivateEye architecture. The client component runs as part of camera service, a trusted module of Android OS. The client and server maintain a persistent TCP connection for as long as the app session runs.

5 IMPLEMENTATION

In this section we provide the implementation details of ePrivateEye. We also discuss the various network configuration used for the evaluation purposes. We implemented ePrivateEye client by modifying the Android Open Source Project (AOSP) version of Android 6.0.1. Our prototype currently runs on a Nexus 5 smartphone. We implemented ePrivateEye server as a standalone desktop application running on Linux, using the OpenCV libraries to perform various computer vision tasks. In our implementation, the version of OpenCV used is 2.4.13.

5.1 ePrivateEye Design

ePrivateEye Client: The client module has three primary responsibilities. The first is to receive frames from the *camera service*, and send them to the server for processing. The client maintains a configuration file which contains the server's IP and port. When ePrivateEye is first instantiated on the device, it opens 3 persistent TCP connections with the server. As soon as the client receives a frame, it places the frame on a queue, *InQueue*, for processing. At this point, a thread with a dedicated TCP connection retrieves the frame from *InQueue* and sends the frame to the server for processing. The second responsibility is to apply the requisite mask to the frame once a response has been received from the server, and to enqueue it onto another queue that the client maintains, *OutQueue*. Upon being enqueued in *OutQueue*, the frame for which the mask information has been received is dequeued from *InQueue*. The client's final responsibility is to communicate to the *camera service* that the processed frame is now ready to be sent to user-level apps.

ePrivateEye Server: The server module runs an event-driven, desktop version of the original *PrivateEye*. The primary responsibility of the server is to perform the computationally expensive marker detection algorithm. The server receives the frame from the client and immediately begins looking for any marked areas. If the server detects a marked region, it sends the corresponding bounding box coordinates which enclose the marked region to the client. Otherwise, the server returns a null response for the frame.

5.2 Server Configurations

Both the client and server modules are designed to be completely portable. This allows for us to construct varying network configurations around the client and server for evaluation purposes. Within our work, we consider three different network configurations running ePrivateEye, and compare the performance against the original PrivateEye running within a local configuration on the mobile device. Because these configurations are applicable to an enterprise setting, we assume that all transmitted data never leaves the confines of the network, and thus remains private from the public internet.

Edge: The *edge* configuration consists of ePrivateEye running on an Access Point (AP). We configured a laptop with a 1.3 GHz Intel Core M CPU and 8 GB of RAM as the AP. The mobile device connects to the AP in order to establish a bridge to the internet. In doing so, the mobile device is able to directly access ePrivateEye. In this configuration, the client and server are separated by a single hop, and communicate using the 802.11ac wireless protocol, with a maximum possible transmission bandwidth of 433 Mb/s.

Building: The *building* configuration consists of ePrivateEye running on a server connected to the same wireless router as the mobile device. The server has the same specs as those described in the *edge* configuration. In this configuration, the mobile device and server are within close proximity of one-another and communicate over the same network using 802.11n, with a maximum bandwidth of 300 Mb/s.

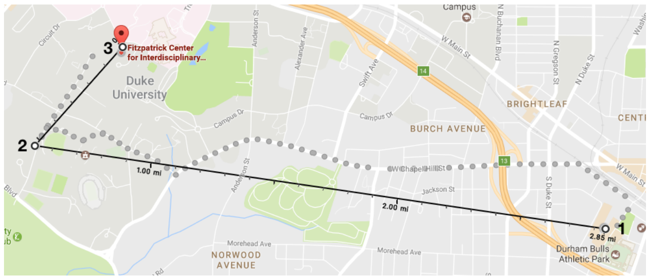

3
Fitzpatrick Center for Interdisciplinary...
Duke University
2
1.00 mi
2.00 mi
2.85 mi
1
Durham Bulls Athletic Park

Figure 4: Metro configuration: The AP resides within the American Tobacco Campus (1). The AP connects to a Cisco 4500X SDN switch located within ATC, and transits across fiber to an Arista 7050 SDN switch in TelCom (2). The switch transits once more over fiber to a final Cisco 4500X SDN switch located within Fitzpatrick East datacenter, which connects via fiber to a server located in the same datacenter that contains a Telsa P100 GPU (3). The total line distance between the AP and server is 2.85 miles. The ping time between the two is 0.5ms.

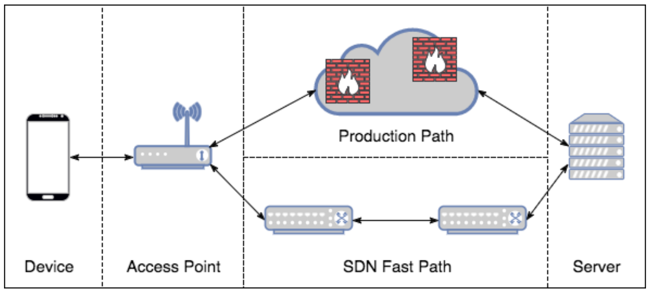


Figure 5: SDN configuration. The top path represents the typical production path through the network, while the bottom path represents the SDN path which traverses two Cisco 4500X and one Arista OpenFlow switches. The choice of path is selected through a controller not pictured in this diagram.

Metro: The *metro* configuration consists of ePrivateEye server and the client being separated across a metropolitan area. The server is equipped with 6 Intel Xeon Processor 2.10 GHz v4 CPUs with 128 GB of RAM. As can be observed in Figure 4, the total distance between the client and server is more than 2 miles. Because of this large distance, we utilize Duke's SDN infrastructure to maintain control over the path through which the communication must occur. In this configuration, the mobile device connects directly to the AP, and the AP connects to the server across a path primarily composed of fiber. Because the transmission time across fiber is approximately two-thirds of the speed of light [21], this line of communication offers the fastest possible path between the AP and server. The maximum bandwidth between the AP and server is 1 Gb/s and has an approximate ping time of 0.593 ms.

In order to ensure maximum bandwidth and minimal latency, we re-use the previously described AP configuration to create a bridge between the mobile device and server. In this configuration, the AP uses a Belkin USB-C Ethernet adapter to interface with the SDN infrastructure. The mobile device connects to the AP using the 802.11ac wireless protocol. The AP serves as a bridge between the mobile device and remote server. As a result of the control of the forwarding elements afforded by the SDN, the AP and server appear to be separated by only a single hop. The maximum possible bandwidth between the server and client is 433 Mb/s. As a result of the near-instantaneous transmission between the AP and server, however, this configuration allows for similar performance as if ePrivateEye was running on the AP.

SDN Fast Path: We leverage Duke's SDN infrastructure to construct a *fast-path* between the AP and server, eliminating any unpredictable sources of latency that might have been incurred along the normal production path. In a production environment, SDN would provide the ideal setting for deploying ePrivateEye because of its ability to create on-the-fly firewall policies, and because it can provide a trusted, fast-path connection between the client and server. Utilizing Duke's SDN allowed us to construct a path that is primarily fiber and consists of the following hops: (1) The Nexus 5 connects to the AP using 802.11ac which creates a bridge to the SDN infrastructure across a Belkin USB-C Ethernet adapter. (2) The Ethernet adapter connects to a Cisco 4500X OpenFlow switch. (3) The first switch connects to a second OpenFlow switch (an Arista 7050) via a DWDM. (4) The second switch connects to a third OpenFlow switch (another Cisco 4500X) located in one of Duke's primary datacenters. (5) The third OpenFlow switch is connected via fiber to a Tesla P100 GPU-enabled server running the ePrivateEye server software. Please refer to Figure 4 for an approximation of this route. The connection between the AP and first OpenFlow switch is a standard copper Ethernet cable, but all remaining links in the topology are using fiber optic cable.

As can be observed in Figure 4, the line distance between the AP and server is approximately 2.85 miles, while the ping time between the two is approximately 0.5ms. As a result of the fiber links and the programming of the forwarding elements, the *fast path* very nearly gives the impression that the AP is connected directly to the server. Since the mobile device connects directly to the AP, however, this configuration, in reality, contains a total of 5 hops between the device and the server.

Figure 5 shows both the typical network configuration, and the SDN configuration for the paths between the AP and server. When the packets traverse the production path, they must be processed by firewalls and other network elements that can add non-deterministic latency to the communication stream. Using Duke's SDN, these sources of unexpected latency can be bypassed through the programmatic control of the forwarding path between the AP and the server. This ensured that our data took a path that resulted in minimal variance for our measurements.

6 EVALUATION

ePrivateEye uses the same computer vision techniques that are used in the existing system, PrivateEye. ePrivateEye is designed to provide better performance and realtime app experience to a user. Therefore, we focus our evaluation on the performance aspect of the system under various settings. We, specifically, want to answer the following:

(1) Can ePrivateEye achieve real-time frame delivery to an app?
(2) Can ePrivateEye continue to provide reliable recognition of marked regions?
(3) What is the cost of running ePrivateEye on a resource constrained smartdevice?
(4) How does increase in number of marked regions in the camera view affect ePrivateEye's performance?
(5) What are the network characteristics required to support smooth functioning of ePrivateEye?

To answer the first two questions, we prepared a video benchmark with typical camera settings and camera movements. We used these videos to compute frames per second (fps) delivery, precision, and recall with which ePrivateEye reveals public regions. Next, we measured the power consumption, CPU usage, and memory consumption to show the impact of running ePrivateEye on a smartdevice's resources. Finally, we evaluated the impact of multiple marked regions on the scalability of our system.

To understand the network support required for a good performance, we investigated the *metro* server configuration where the server and client are separated by a distance of 2.8 miles. Specifically, we measured the impact of network delays and packet losses on the frames per second delivery. Further, we investigated the case of cloud infrastructure where ePrivateEye server is run on Amazon Web Server (AWS).

6.1 Experiment Setup

Video Benchmark: To carefully study the accuracy and impact of ePrivateEye for different camera settings and motions, we developed a simple video benchmark. We recorded several videos, each 20 seconds in length, at a resolution of 1280×960 at 30 fps using a Nexus 5 smartphone. The average bitrate for each video was computed to be 9321.33 kb/s.

We focused on three different two-dimensional regions: whiteboard, presentation, and laptop screen. We marked a public region on each two-dimensional region and recorded the video under the following three camera motions: (1) *Still*: the camera was steady and remained focused on the marked region for the entire video, (2) *Spin*: the camera rotated while remaining focused on the marked region, and (3) *Scan*: the marked region first moved out of the camera view and then came back in. The three camera motion *still*, *scan* and *spin* simulate the scenario of still photography, video recording, accidental capture of public regions, and changing orientation of the recording device. We recorded videos for all the three two-dimensional regions under the three different camera settings. As a result of space constraints, we only show results obtained for public regions marked on whiteboard.

To test ePrivateEye, we modified the camera service to read the pre-recorded video files and load them into the memory. When a camera frame is delivered from the camera sensor to the camera service, the camera service replaces the frame data with the data coming from the pre-recorded videos and passes them to the apps. For the app, the frames appear to arrive directly from the camera sensor. Using this setup, we measure precision, recall, and performance of ePrivateEye under different scenarios.

ePrivateEye Setup: Our goal is to show the performance of the edge-computing based system, ePrivateEye, under each of several different scenarios and, at the same time, present a comparison with the local-computing based system, PrivateEye. Therefore, we configured ePrivateEye server in different ways to simulate those scenarios: (1) On-device: where the server runs on the smartphone; this simulates the previous system PrivateEye, (2) Edge: where the server runs on an access point to which the smartphone (ePrivateEye client) is connected directly, (3) Building: where the ePrivateEye server is reachable via a router, and (4) Metro: where the server and client are separated physically by a large distance and connected through different networks. The details of the various server configurations are previously described in Section 5.2.

There is one subtle difference between PrivateEye and *on-device* configuration of ePrivateEye. As noted earlier in Section 3, PrivateEye achieves higher fps by running primarily in tracking mode, with periodic transitions to detection mode. We must note, however, that in our *on-device* configuration ePrivateEye performs detection on every frame; we did so in the interest of having the code remain invariant while performing comparisons using differing server configurations.

6.2 Frames per Second

To compute fps we set the *still*, *spin*, and *scan* videos to loop, replay them for 30 seconds, and record the times at which each frame is finished processing. We then process the logs to compute the rate of frame delivery for each configuration.

As can be observed across Figures 6, 7, 8, ePrivateEye significantly outperforms PrivateEye, achieving a median frame rate of $\approx$ 30 fps across each remote configuration. Because the

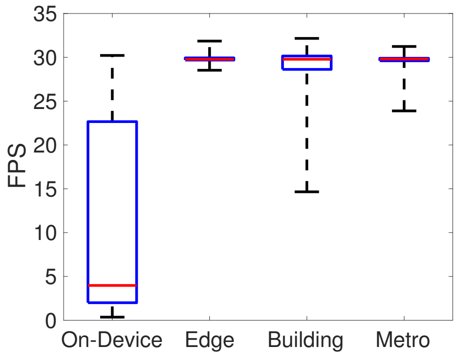


Figure 6: FPS results for the *still* camera movement.

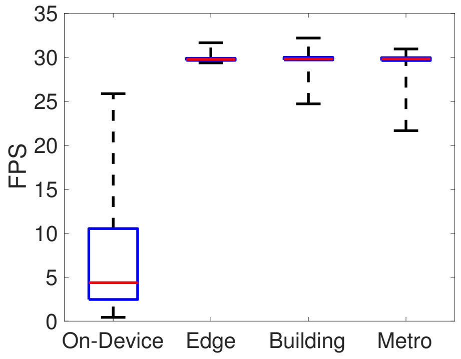


Figure 7: FPS results for the *spin* camera movement.

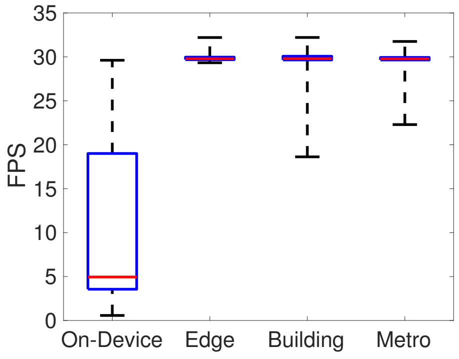


Figure 8: FPS results for the *scan* camera movement.

processing time for a single image on the server was ≈ 6ms, ePrivateEye can handle frame rates exceeding 30 fps. In comparison, the *on-device* configuration achieves a median frame rate of ≈ 5 fps, equivalent to an 83.33% reduction in performance. Although PrivateEye has proven to have a frame rate of ≈ 20 fps, the detection of marked regions in consecutive frames causes the performance to deteriorate. ePrivateEye offers a significant improvement in quality of service over that provided by PrivateEye, by running in real-time with no noticeable lag or delay.

While the median fps for all the camera motion settings are around 5 fps, the *on-device* configuration delivers some frames at ≈ 18 fps for *still* and *scan* camera motion settings. In case of *still* the marked region is in focus, which eases the task of detection for the software. In case of *scan*, since the view is constantly changing, the frame data can become blurry, and the software does not always have as many distinct features in the image to process. This results in occasional fast processing of the frame and faster delivery of the frames. In the *spin* case, the object stays in the focus throughout the video, but the orientation changes constantly. This causes the detection to run on every frame with many features to process, and, as a result, never reaches a high rate of frame delivery. This observation is in agreement with the frames per second evaluation of PrivateEye [18].

Between the remote configurations, the *edge* configuration produces the most consistent results with the lowest amount of standard deviation. As this configuration establishes the shortest path between the client and server, these results are expected. The *metro* configuration produces the second most consistent results, as can be observed in Figures 7 and 8. Although the standard deviation for the *building* configuration is higher than the other two configurations, it still achieves a median frame rate of ≈ 30 fps.

The reason that the *metro* configuration has a more consistent performance than the *building* is because of two reasons. First, the SDN establishes a *fast path* between the AP and server which has minimal overhead due to the selection of the path. Second, within the *building* configuration, the mobile device and server both utilize the 802.11n wireless, whereas within the *metro* configuration, the mobile device communicates using the 802.11ac protocol, which has a higher maximum bandwidth. Further, because the mobile device and server are both connected to the router over the wireless connection, there is a higher probability for packet loss and latency to occur. This shows the importance of using SDN – even though the client and server are separated by a few feet in the *building* configuration, the *metro* configuration, where the client and server are separated by 2.85 miles, produces superior performance results.

Overall, these results show that realtime performance can be achieved across each remote configuration of ePrivateEye, with the *edge* offering the least deviation from the optimal performance of 30 fps.

6.3 Precision and Recall

To compute precision and recall, we divide the prerecorded videos into pixel cells of size 5×5. We manually mark all the cells inside the marked region as *public* and cells outside the marked region as *private*. Our system should reveal *public* cells and block *private* cells. We consider a pixel cell from a frame to be blocked if none of it is visible in the corresponding frame processed by ePrivateEye; otherwise we consider the cell to be revealed by our system. This is a strict enforcement as we assume that a partially visible cell can have enough information to reveal the entire cell.

Formally, for a given frame, let C_t and C_r be the number of cells marked public and number of cells revealed by the system, respectively. Then we define precision and recall for that frame as follows:

$$precision = \frac{|C_t \cap C_r|}{|C_r|} \qquad recall = \frac{|C_t \cap C_r|}{|C_t|}$$

While precision represents how secure is our system, recall represents how usable the camera frame is to an app.

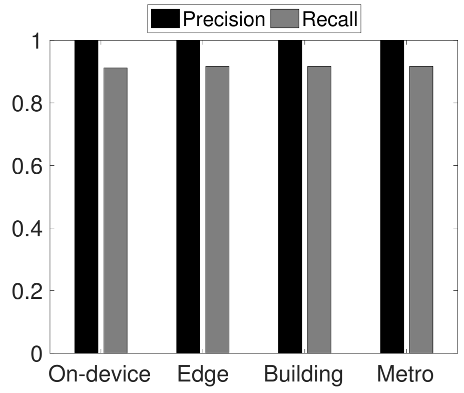


Figure 9: Precision and recall for *still* camera motion.

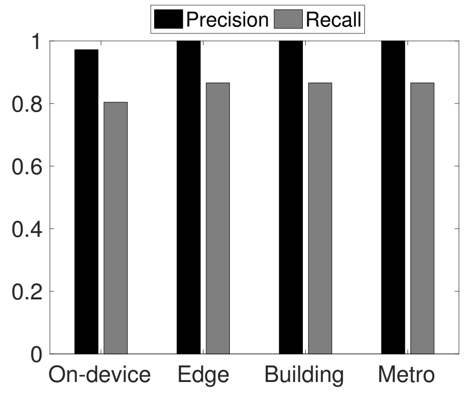


Figure 10: Precision and recall for *spin* camera motion.

Figures 9, 10, 11 show that ePrivateEye achieves a 100% precision for all camera motion settings. This indicates that ePrivateEye does not let any sensitive information through to any app. Further, the evaluation reveals that precision and recall results remain constant across each network configuration running ePrivateEye. This is an expected outcome as change in network settings has nothing to do with the detection of

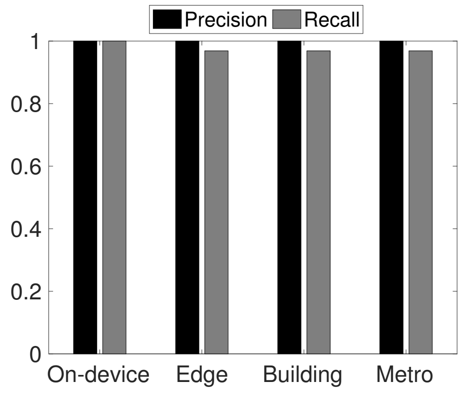


Figure 11: Precision and recall for *scan* camera motion.

marked regions as long as the frame data doesn’t get modified while in-transit. This indicates that running ePrivateEye in any remote configuration will produce exceptionally strong precision results.

On-device configuration provides a median precision of 0.96 in case of *spin* camera setting, while achieving a 100% precision for *still* and *scan* settings. In *still* camera setting, the camera is focused on the marked region which helps in picking up the features to perform detection effectively on the device.

The precision for *on-device* configuration under *spin* camera setting drops slightly. Under this camera motion, although, the marked region stays in focus of the camera, its orientation changes constantly which makes the detection of key features slightly difficult. However, for the same camera setting, all the remote configurations achieve higher precision. We suspect that this difference in precision values across configurations comes from the OpenCV libraries used by the ePrivateEye. The OpenCV libraries undergo different optimizations for them to work on resource constrained ARM-based platform. As the remote configurations have significantly stronger processing power, they are able to take full advantage of the functionality that OpenCV has to offer.

The cost of higher precision for ePrivateEye comes with the tradeoff of decreased recall performance. In comparison to PrivateEye, ePrivateEye has slightly weaker recall results. Because recall is indicative of usability, the remote configurations of ePrivateEye underperforms in this category. However, the performance does not suffer too much as ePrivateEye still achieves a minimum recall of 82%. Overall, although the recall performance decreases when using ePrivateEye, the perfect precision coupled with the high rate of frame delivery as observed in the previous section, makes ePrivateEye the superior implementation in terms of both security and QoS.

6.4 Resource Usage

We wanted to investigate the impact of running ePrivateEye on a mobile device's resource usage. To do this, we designed an experiment where we recorded a 30 second video containing one marked public region. We set the camera setting to be *spin*. We ran this prerecorded video 5 times through ePrivateEye and measured CPU usage (in percentage), memory usage (in MB) and power consumption (in mW). For this experiment, we fixed the smartphone's brightness level to 50% and rebooted the device between trials. During all the trials, we kept the wireless radio active.

We took measurements every 100 ms using Trepn [14], a performance monitoring tool developed by Qualcomm. Trepn reports very accurate measurements for the devices that come with Qualcom's snapdragon processor and Nexus 5 is one of them. We used this app and recorded the CPU usage, memory consumption and power consumption.

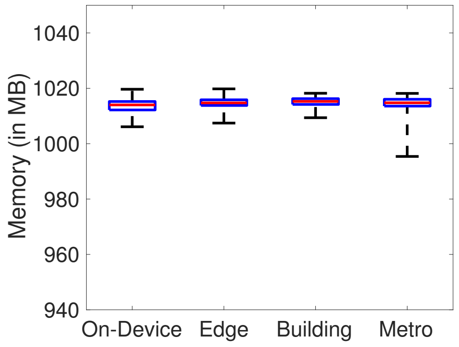


Figure 12: Memory usage for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

In case of memory consumption, offloading compute intensive processing has no impact. The ePrivateEye client maintains a pre-allocated memory to hold a constant number of frames that needs to be processed by the server component. Due to this, we do not see any significant change in memory consumption (Figure 12).

Figure 13 shows that ePrivateEye server running on the device causes a higher median CPU load (≈ 50%) than that of the case when the server runs remotely (≈ 40%). Interestingly, even after offloading the computationally intensive part of processing, the difference between the CPU load is not very high. The reason is that in case of remote processing (*edge* , *building* and *metro*), the CPU stays active as it can process significantly more frames (real-time processing). As shown earlier, ePrivateEye processes 500% more frames per second when it is offloaded.

Similarly, while the power consumption across all the configurations appear comparable (Figure 14), the real gains can only be understood when the results are looked with the number of frames processed per second. Figure 14 and Figure 7 show that on-device processing consumes ≈ 144 J while delivering frames at a median rate of 5. On the other hand, processing frames on a remote server, ePrivateEye consumes ≈ 156 J while delivering frames at a median rate of 30.

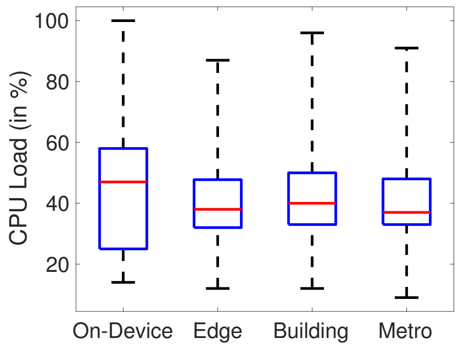


Figure 13: CPU load for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

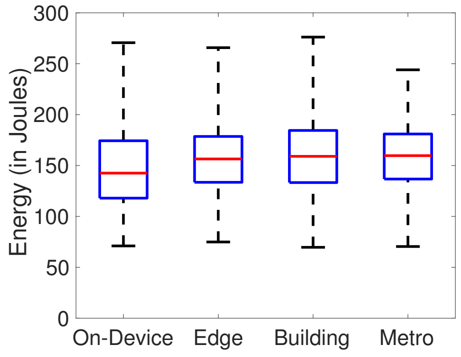


Figure 14: Power consumption for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

6.5 Scalability

To measure scalability, we expand our video benchmark to include three additional 20 second videos containing 1, 2, and 3 marked regions respectively under the camera setting *still*. These videos were recorded at the previously mentioned resolution and frame rate on a Nexus 5 smartphone. The purpose of adding more marked regions to the frame is to increase the number of computations required to run the detection algorithm. As more regions are introduced, the computational complexity increases which places a larger load on the CPU on a per-frame basis. When measuring the performance, we replay these videos for their full duration and record the processing time for each frame.

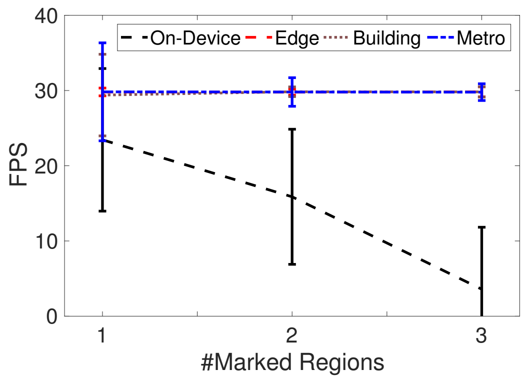


Figure 15: (color) Median rate of frame delivery under different configuration of ePrivateEye server.

The evaluation results for PrivateEye and ePrivateEye can be observed in Figure 15. As is expected, the fps performance for PrivateEye decreases as the number of marked regions increases. The median frame rate for the video with 1 marked region is 20 fps, while the median performance for three marked regions is 5 fps, corresponding to a 75% performance decrease. As the number of marked regions increases, PrivateEye struggles to scale and the usability suffers.

In comparison, the plots for ePrivateEye displayed in Figure 15 show that the median frame rate across each network configuration remains at 30 fps. This shows that performing the processing on the server does not add any additional latency as the number of marked regions increases. For the video with 3 marked regions, the median performance for ePrivateEye offers a 500% improvement over the *on-device* configuration (30 fps v. 5 fps). These results show that ePrivateEye can continue to offer realtime processing rates as the complexity of the operation increases.

7 CASE STUDY: METRO

We wanted to investigate the impact of poor network conditions on the performance of ePrivateEye. Because the *fast path* configuration represents a near optimal connection between the AP and server, we wanted to measure the performance of ePrivateEye under less than ideal conditions. To create these conditions, we induce both latency and packet loss on the egress connection of the server. We then leverage the use of the SDN to eliminate all external sources of latency and packet loss, in order to ensure that the only source of the performance decay is from the ingress location. This allows us to construct a *slow path* and *lossy path* across the *metro* network configuration.

We introduce latency and packet loss to the network using `netem`, a Linux package designed to perform wide area network emulation [9]. We construct the *slow path* by adding latency to the network in increments of 10ms, iterating from 0 to 70ms. We construct the *lossy path* by adding random packet loss to the network in increments of 1%, iterating from 0 to 10%. We perform replay using the video containing 2 marked regions, and collect fps processing data for each path configuration.

7.1 Impact of Packet Loss

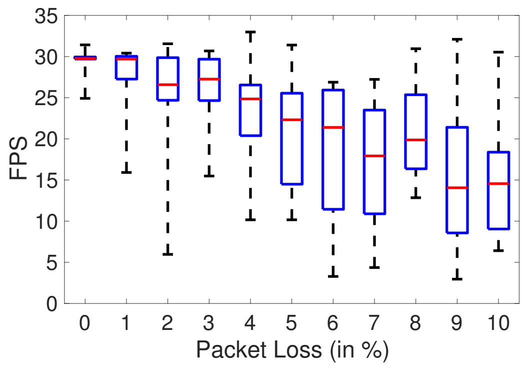


Figure 16: Impact of packet loss on the rate of frame delivery.

Because the client and server maintain several TCP connections, adding packet loss on the ingress link causes the client to start sending retransmission packets. As the loss increases from 1 to 10%, the amount of retransmission traffic increases proportionally. This has the effect of adding latency to the application since the server will not be able to process the frame until it arrives completely. Figure 16 shows the performance results for ePrivateEye runinng across the *lossy path* in the *metro* network configuration. As can be observed, the impact of packet loss on the network can be observed almost instantaneously as the loss rate increases from 1 to 2%. This causes the median fps rate to drop from 30 fps to 26 fps. At 5% packet loss, the performance of ePrivateEye has dropped by 25% from 30 fps to 22.5 fps, and by 10% packet loss, the performance has dropped by 50% to $\approx$ 15 fps. The results indicate a nearly linear relationship between packet loss and number of frames delivered per second. Still, ePrivateEye is able to outperform the *on-device* configuration all the way up to 8% packet loss, and stays competitive through 10% packet loss.

In addition to the median fps decreasing, as the amount of packet loss increases, the variance between fps greatly increases. This has the visual effect of adding a large amount of judder to the experience of using the camera app. Since packet loss is not uniform, this creates periods of both low and high fps causing the camera app, at times, to appear laggy before returning to realtime.

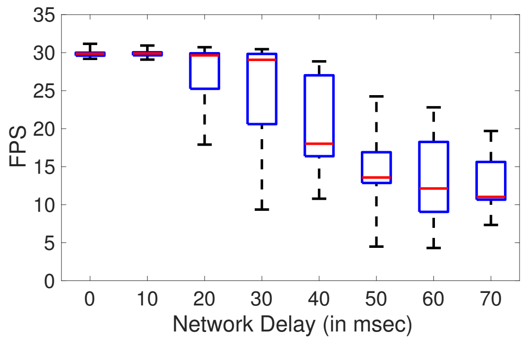


Figure 17: Impact of network delay on the rate of frame delivery.

7.2 Impact of Network Delay

Figure 17 shows the performance results for ePrivateEye runninng across the *slow path* in the *metro* network configuration. As can be observed, ePrivateEye maintains a median performance of 30 fps as the latency increases from 0 to 30 ms. When the latency increases from 30 to 40 ms of delay, the median performance drops by 42% to ≈ 17.5 fps. The reason for this drop is due to the processing rate of frames within the *camera service*. Because the *camera service* receives frames at a rate of 30 fps from the camera sensor, ePrivateEye has approximately 33 ms to perform the remote processing before the next frame is received. As long as the client and server are able to complete the data exchange within this time frame, the performance will not suffer. As soon as the latency eclipses this window, the performance of ePrivateEye becomes dictated by the latency of the network.

Although the performance starts to decay after this point, ePrivateEye is able to produce decent performance results. With 50 ms of delay, ePrivateEye still performs at a median rate of ≈ 15 fps, which is competitive with the *on-device* performance as can be observed in Figure 15. This result show that even under networks with high latency (≈ 30 ms), ePrivateEye is able to produce strong performance results.

8 ePrivateEye IN THE CLOUD

PrivateEye is designed to protect sensitive information from being accidentally captured by the camera apps. ePrivateEye extends the original system to provide realtime performance without compromising the privacy guarantees. Therefore, it is of utmost importance that the edge computing resource used for running the ePrivateEye server is trusted. We envisioned a futuristic scenario, where a trusted cloud service provider aims to provide the benefits of ePrivateEye to the masses by running the server component on its secured cloud infrastructure. We are interested to see the rate of frame delivery in such a scenario.

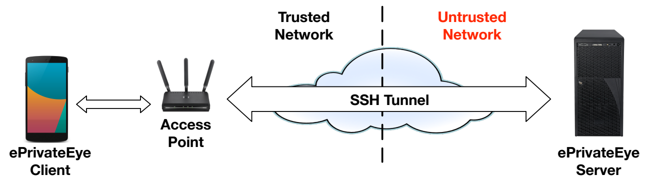


Figure 18: The secure channel setup to provide connectivity between the ePrivateEye client on Duke's trusted network and ePrivateEye server on an instance of AWS server.

For this, we used the AWS cloud infrastructure and configured the ePrivateEye server on an instance of AWS machine. Ideally, when the camera data leaves the smartdevice for processing on a cloud infrastructure, it should be encrypted. To emulate this scenario we setup Nexus 5 (client) to connect to an AP (within Duke University's network) and established a SSH Tunnel from the AP to the AWS server (Figure 18). We measured the rate at which the frames were delivered to the app for a video with one marked region on a whiteboard under all the three camera motion setting.

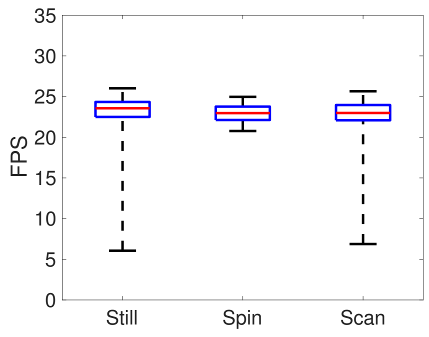


Figure 19: FPS achieved by configuring ePrivateEye server on an instance of AWS server.

Figure 19 shows that under the described setup the median rate at which frames are delivered is 23. This is due to the fact that SSH tunnel adds an additional computation burden of encryption and decryption. We noticed significant aberrations in processing delay during the *still* and *scan* recordings, showing how the additional network overhead can be unpredictable at times. Overall, this is a very encouraging result as it shows that ePrivateEye can give an acceptable performance under an additional layer of security.

9 DISCUSSION

Impact of Offloading: The evaluation results of ePrivateEye show that by offloading to the edge, the performance of the original PrivateEye can be vastly improved. In this configuration, the system achieves a frame rate of 30 fps and a precision of 100%. At the same time, the CPU load decreases by 5% from running ePrivateEye *on-device*, while the memory consumption remains unchanged. The only performance limitations include a slight increase in power consumption of 8.3%. The primary improvement over PrivateEye can be observed by ePrivateEye's ability to handle scaling. ePrivateEye maintains a high level of performance while tracking multiple marked regions, whereas PrivateEye's performance degrades significantly when there's more than one region. With such a performance, ePrivateEye enables real-time camera apps to function smoothly while providing high privacy guarantees.

Managing Network Conditions: The network configuration *edge* is well suited for home network, while *building* and *metro* configurations can be adopted in an enterprise setting. By varying the network configurations, we have explored the network conditions that might affect the performance of ePrivateEye. The results show that regardless of the offloading configuration, ePrivateEye will provide the performance guarantees to implement and scale the system. The lossy network evaluation shows ePrivateEye can sustain a packet loss up to 8% and a network delay up to 40 ms while delivering frames at the median rate of 20 per second. To ensure optimal performance under varying conditions, SDN offers a viable solution by providing greater control over path selection.

Deep Learning: While the *edge* configuration offers the best performance, the *metro* configuration offers the most promising results. The remote server within this configuration presents the possibility of running heavier computations that require GPUs to run e.g. deep neural networks. The size of the input on the GPU represented an insignificant load, and was easily handled. Advanced computer vision techniques coupled with deep learning can classify several classes of objects or shapes, but require this level of computing to operate at real-time speeds. Because the marker detection is completely run on the server, classifiers can be added or updated without changing anything on the client side.

10 RELATED WORK

Mobile Offloading: MAUI [7] and LEO [12] make runtime scheduling decisions about how to split processing between on-device and remote execution. Both systems solve an Integer Linear Program (ILP) that encapsulates the context in which the system is running, to determine how to best distribute processing. Rather than performing a runtime decision, ePrivateEye offloads the most computationally expensive portion of the image processing pipeline. This fixed offloading approach avoids the complexity of synchronizing client and the server. Further, based on the quality of network, ePrivateEye can be quickly reconfigured to switch between on-device and remote processing.

Kahawai [8], a platform for high-quality gaming, performs collaborative rendering between server and client. Within this work, both the client and server execute the application, and if the server finishes before the client, the client will stop processing and incorporate the completed results from the server. ePrivateEye avoids the redundancy of performing the remote and local processing concurrently due to the high overhead imposed by the local processing.

CloneCloud [6] transforms mobile applications into a distributed system by selecting a single thread running on the device to be executed in a clone running remotely. Because the most computationally expensive component of ePrivateEye is known ahead of time, this section is offloaded entirely rather than having to make a runtime decision.

Network Optimization: Systems relying on remote processing are dependent on network conditions. IC-Cloud [20] predicts network conditions using signal strength and historical information to make lightweight offloading decisions. Whereas IC-Cloud makes internal decisions to combat poor network quality, we show that ePrivateEye maintains a strong performance across varying network conditions.

SDNs have become ubiquitous in several enterprise and campus networks, and provide more control over the data flow. ECOS [11] utilizes software defined networking to control mobile application offloading with a focus on ensuring privacy, fair resource allocation, and enforcing security constraints. ECOS is implemented as an application running within a NOX controller, and communicates with mobile clients seeking to perform application offloading with certain privacy and performance requirements. While ECOS primarily focuses on security, we utilize SDN within our network to minimize network latency.

11 CONCLUSION

This paper presents ePrivateEye, an edge-enabled version of PrivateEye which offloads the computationally intensive marker detection to an edge server. ePrivateEye outperforms the existing system and achieves realtime performance (30 fps) while providing high security (precision 1) and acceptable app usability (recall ≥ 0.86) at a small cost of increased energy consumption (8%). In addition, we perform case studies to show that ePrivateEye can be adopted for large metropolitan areas as it can resist packet losses up to 10% and network delays up to 30 ms.

ACKNOWLEDGMENTS

The authors would like to thank the Duke's Office of Information Technology for supporting this work and helping us in setting up the SDN and associated GPU environment for evaluating the "metro" area scenarios; we are most especially grateful to: Scott Steagall, Mike Newton, Carl McMillon, Ken Foster, Travis Sherron, Tom Nohs, Keith Gentry, and Charley Kneifel.

REFERENCES

[1] P. Bahl. The emergence of micro datacenters (cloudlets) for mobile computing, 2015.

[2] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proc. of MobiSys*, 2007.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC ’12, pages 13–16, New York, NY, USA, 2012. ACM.

[4] J.-Y. Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[5] J. Canny. A computational approach to edge detection. *TMAPI*, 1986.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[8] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135. ACM, 2015.

[9] T. L. Foundation. Netem. [Online; accessed 22-Apr-2017].

[10] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[11] A. Gember, C. Dragga, and A. Akella. Ecos: leveraging software-defined networks to support mobile application offloading. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 199–210. ACM, 2012.

[12] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, number CONFCODE, pages 320–333. ACM, 2016.

[13] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’15, pages 137–150, New York, NY, USA, 2015. ACM.

[14] Q. T. Inc. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler. [Online; accessed 7-Dec-2015].

[15] K. Lee, D. Chu, E. Cuervo, J. Kopf, A. Wolman, Y. Degtyarev, S. Grizan, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. *GetMobile: Mobile Comp. and Comm.*, 19(3):14–17, Dec. 2015.

[16] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. MobiSys, 2011.

[17] N. Raval, A. Srivastava, K. Lebeck, L. Cox, and A. Machanavajjhala. Markit: Privacy markers for protecting visual secrets. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 1289–1295. ACM, 2014.

[18] N. Raval, A. Srivastava, A. Razeen, K. Lebeck, A. Machanavajjhala, and L. P. Cox. What you mark is what apps see. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 249–261. ACM, 2016.

[19] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.

[20] C. Shi, P. Pandurangan, K. Ni, J. Yang, M. Ammar, M. Naik, and E. Zegura. Ic-cloud: Computation offloading to an intermittently-connected cloud. Technical report, Georgia Institute of Technology, 2013.

[21] A. Singla, B. Chandrasekaran, P. Godfrey, and B. Maggs. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 1. ACM, 2014.

[22] S. Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, 30(1):32–46, 1985.