Securing Recognizers for Rich Video Applications

Christopher Thompson
University of California, Berkeley
cthompson@cs.berkeley.edu

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

ABSTRACT

Cameras have become nearly ubiquitous with the rise of smartphones and laptops. New wearable devices, such as Google Glass, focus directly on using live video data to enable augmented reality and contextually enabled services. However, granting applications full access to video data exposes more information than is necessary for their functionality, introducing privacy risks. We propose a privilege-separation architecture for visual recognizer applications that encourages modularization and least privilege—separating the recognizer logic, sandboxing it to restrict filesystem and network access, and restricting what it can extract from the raw video data. We designed and implemented a prototype that separates the recognizer and application modules and evaluated our architecture on a set of 17 computer-vision applications. Our experiments show that our prototype incurs low overhead for each of these applications, reduces some of the privacy risks associated with these applications, and in some cases can actually increase the performance due to increased parallelism and concurrency.

1. INTRODUCTION

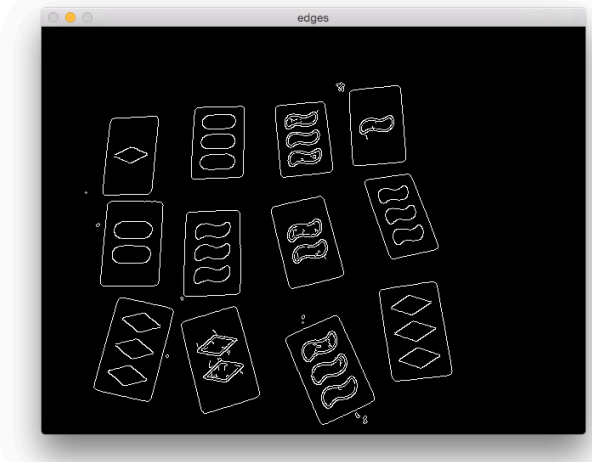
Sensor-rich, contextual, and augmented reality computing is becoming more prevalent with devices like Google Glass [10], Meta [2], Kinect [16], and an increasing number of smartphone applications, such as Layar [1]. In response, Jana *et al.* [12] introduce a privilege separation architecture for such applications that protects privacy by segregating sensor data processing into a separate module. A recognizer implements a single task on sensor data: a recognizer "recognizes" and outputs certain image features the application wants access to. Instead of giving the application full access to the sensor data, the application receives access only to features extracted by the recognizer. For instance, a face-detection recognizer might analyze video frames and output the location of all visible faces. Figure 1 shows an example of the input and output of a edge detection recognizer.

SPSM'16, October 24 2016, Vienna, Austria

ISBN 978-1-4503-4564-4/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2994459.2994461

(a) The original video frame



(b) The extracted contours drawn on a blank background.

Figure 1: Detecting the contours of edges is a good example of a computer vision recognizer. From a full video frame, only the contour points of the edges are extracted. The detected edges might be used directly by the application or could also be used as input to other recognition tasks.

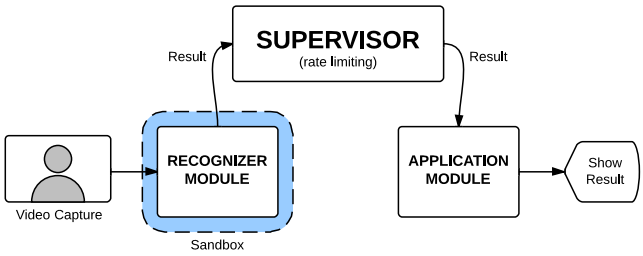


Figure 2: Our recognizer architecture. The recognizer module has access to the camera but can be securely sandboxed to prevent network and filesystem access. The application module receives the results of the recognizer through the supervisor proxy, which restricts how much data the recognizer can send.

Previous work focused on an architecture where applications are limited to a set of OS-provided recognizers [12]. However, the space of features that applications might want to be able to extract from live video is endless. As soon as an application wants to deviate from the OS-provided recognizers, this approach becomes insufficient. Perhaps the application needs to implement novel, unexpected functionality, or it has certain performance needs that the system's algorithms do not provide.

We focus on applications that use potentially sensitive sensor data, such as live video feeds, motivating the design of an architecture that can protect users from applications that, while non-malicious, may not be 100% trustworthy. This is particularly relevant for smartphones, where the incidence of malware has been very low [14]: applications that are accidentally over-privileged [8] or misuse personal information and identifiers [7] are far more common than malicious applications. These applications might not intend to leak sensitive data or perform malicious actions, but might end up doing so inadvertently.

We propose a new architecture to address these challenges, illustrated in Figure 2. Our recognizer architecture focuses on allowing novel recognizers that the system designers may not have anticipated, while still restricting access to raw sensor data and encouraging security-sensitive application design. Our architecture separates the computer vision tasks into a distinct "recognizer" module, and restricts the bandwidth at which the recognizer module can communicate with the main application module. The bandwidth restriction encourages developers to place any code requiring access to the raw sensor streams inside the recognizer module, and to only extract the minimal features required for the application's functionality. Thus, our architecture encourages privilege separation and least privilege of the components. This separation allows the system to sandbox the recognizer code, preventing actions that could leak information, such as network access or writing to the filesystem. This design allows general computation on sensor streams, while encouraging and assisting application developers to adopt a more private and secure design.

Consider an augmented reality application that allows a user to sculpt virtual objects with their hands. A conventional implementation of this application would request full access to the video in order to extract hand and finger positions so it can recognize gestures and interaction with the virtual sculpture. Because this application receives full access to video feeds, if this application were to be compromised, it could be used to spy on the user and others in the vicinity. Essentially, if any application that does computer vision is compromised by an attacker, it becomes a remote access trojan (RAT).

Let's look at how this sculpting application would work in our privilege-separation architecture. The recognizer module would analyze each frame to extract hand and finger positions, angles, and depths, and output only these key features. The core application would show the UI, communicate with network services, and use the outputs from the recognizer to manipulate the virtual sculpture.

This greatly reduces the risk of compromise. Vulnerabilities in the core application cannot be used to spy on the user, as the core application does not have access to the video feed. Also, our architecture provides partial protection against vulnerabilities in the recognizer module: such vulnerabilities might allow the attacker to extract information about the user's surroundings (for example, decoding and exfiltrating a credit card number, if their credit card is ever held within the field of view of the camera), but the bandwidth limit on the channel between the recognizer and the core application means the compromised recognizer cannot be used to record an embarrassing video and upload it to the attacker.

We implemented a prototype of our recognizer architecture in Python, using the popular OpenCV computer vision library [11]. We constructed a benchmark suite based upon applications from the Darkly corpus [13] (ported to Python) as well as a few other example applications we selected. We evaluated our architecture's performance by comparing the performance of these applications in our privilege-separated architecture, their performance as monolithic Python code, and their original native code performance. We found that our recognizer architecture imposes minimal performance overhead, and in some cases increases performance due to an increase in parallelism and concurrency. We argue that video recognizer applications function well under a bandwidth-constrained channel between the recognizer and the application, and this can greatly reduce the chances of exfiltration or misuse of raw video data.

2. BACKGROUND

Traditionally, a third-party application's access to sensor data has been dependent on user permissions. For example, when an iOS application first requests access to a particular sensor, iOS will display a prompt asking the user to decide whether to grant that request (and then never ask again); Android displays a prior-to-installation manifest outlining the sensor usage that the application wants and will only permit installation if the user agrees to this sensor usage. These types of systems are all-or-nothing (the application either receives full access to that sensor's datastream, or no access at all), do not encourage least privilege, and rely on the user both having the expertise to make an informed decision, and taking the time to make that decision.

In order to tackle these issues, Roesner *et al.* developed a new style of permission granting, using access control gadgets [19]. Their focus was to develop a system that maintained least privilege, while also being non-disruptive to users. They did this by developing a system where permis-

sion granting is built into the actions the user performs with the application: the application gets permission to various resources via the user's interactions with an access control gadget. Although this system does encourage developers to avoid requesting permanent access when it is unnecessary, and gives users the choice to revoke said access, it nonetheless is still all-or-nothing and thus has the potential to provide more information to the application than the application explicitly needs. TaintDroid [6] takes a different approach: information flow control is used after applications get access to sensitive data to determine what sensitive data can be used in what context by the application.

DARKLY [13] combined access control, algorithmic transformation of sensor data, and user audit to provide privacy protection. They also use privilege separation; their system is split into two parts: a trusted local server and an untrusted client library. The trusted DARKLY server is a privileged process that has direct access to the sensors; applications get "opaque references" to all sensitive data, which can only be dereferenced by their modified OpenCV functions, or accessed through special "declassifier" routines, which must be specifically designed. They evaluated their system by testing it on 20 existing OpenCV applications, measuring whether any modification of the application was required, and how much the functionality and accuracy of the applications degraded. We base our testing corpus on this set of applications as a way to evaluate the applicability and performance of our architecture.

Other work has focused on the growth of wearable and ubiquitous cameras. PlaceAvoider [22] is a probabilistic framework used to detect where a photo was taken. It flags photos taken in sensitive locations (such as bathrooms) for user review before they are made available to applications. MarkIt [18] is a privacy marker framework that allows users to mark off parts of a video feed that are sensitive and should not be captured in the future.

Jana *et al.* introduced the abstraction of the "recognizer" [12]. A recognizer takes the raw sensor data as input and outputs higher-level objects (such as a face or motion vectors). Recognizers allow for a fine-grained permission system: applications can request output from a specific recognizer, and this provides a clean way for users to understand what information the application is gaining access to. Jana *et al.* argued for a small fixed set of system-provided recognizers, corresponding to common computer vision tasks they found that met all of their applications' needs (e.g., hand tracking, skeleton detection). However, this fixed set of recognizers does not allow for general computation on the sensor data. Novel applications could easily require new recognizers or customized versions of existing recognizers. For example, an application wanting access to the color temperature of the scene would not be able to function with their fixed set of recognizers. Our work focuses on how to allow for novel recognizers in a secure manner.

3. SECURITY ARCHITECTURE

The goal of our security architecture is to enforce the modularization and least privilege of visual recognizer applications. Such applications should separate their computer vision logic (the "recognizer" logic) from their application logic. The recognizer logic only needs access to the computer vision APIs, the camera, and a limited amount of state, but most filesystem access and all network access can be removed. The application logic has full privileges except it is not given access to the camera. This design is similar to the design of "content scripts" in Chrome's extension architecture [4, 9].

Compared to previous security mechanisms for visual recognizer applications [12], we allow fully general programming of the recognizer, therefore developers can come up with novel recognizers or customized versions of existing ones, instead of having to rely on a fixed set of system-provided recognizers. We focus on a much more generalizable application framework than either DARKLY or system-provided recognizers. Our architecture provides secure access to the live camera, while being easy to use for applications that interact with other libraries and services.

Our design is summarized in Figure 2. The input to the recognizer is a sequence of video frames, and we assume it will do some processing on each frame separately and output something for each one. In particular, we assume it will do some computer vision processing separately on each frame (which is pure computation on this input) and then output some information (e.g., some high-level semantic features) about that frame. The recognizer sends its output to the application, proxied through the supervisor. The supervisor limits the rate at which the recognizer can send data. This serves two functions. First, it prevents a lazy programmer from simply forwarding the entire video stream to the application module, maintaining separation of privilege for the entire application. Second, it limits the amount of information from the camera that could be exfiltrated by the application. Additionally, the supervisor can periodically reset the recognizer module to prevent it from accumulating state about past video frames and exfiltrating it slowly.

Our architecture targets applications which are non-malicious but potentially harmful or exploitable. In particular, we do not directly protect against side channels, although our sandbox implementation indirectly eliminates most useful high-bandwidth channels (which typically rely on network or filesystem resources). Instead, we focus on non-malicious but possibly buggy or careless applications. A careless application might unintentionally expose raw video data to cloud services or third parties (potentially through advertising code). A buggy or poorly implemented application might be compromised and used to access the raw camera feed. Our sandbox architecture protects against all of these threats.

3.1 Implementation

Our supervisor, written in Python, spawns a child process for the recognizer and the application modules and then executes the main functions for each in these new processes.

The supervisor also acts as a proxy for all data sent between the recognizer and the application. The modules communicate over named IPC sockets created by the supervisor using zeromq and passed to the recognizer and application. The supervisor limits the bandwidth of this channel using a token bucket with configurable fill rate and bucket size. To ensure liveness of results from the recognizer, the token bucket uses a deque with a max-length of 2. This max buffer size can cause frames to be dropped but prevents old results from filling up the queue and being delivered to the application after they are no longer contextually relevant. We implemented optional automatic zlib compression of all objects sent over these sockets. The recognizer compresses the

marshaled byte-string and the application decompresses it before un-marshaling.

The application can also communicate settings back to the recognizer through another set of sockets. This channel is limited to JSON dictionaries of settings, to prevent the application module from updating the code of the recognizer module, but is not bandwidth-restricted. Several of the applications in our testing corpus perform such bi-directional communication.

We chose Python and zeromq for ease of implementation and portability. For instance, our implementation could be easily ported to other platforms simply by changing the sandbox implementation. OpenCV and zeromq both have Java and Objective-C bindings, so our design could be easily ported to Android and iOS, two platforms where we see visual recognizer applications becoming more popular, and where fine-grained permission systems already exist. Other vision libraries could be used as well, as very little of our design depends directly on OpenCV.

The recognizer can easily be sandboxed using a system such as seccomp-bpf, to restrict the recognizer module to limited filesystem access and no network access. Our implementation supports the use of either seccomp-bpf on Linux or the OS X sandbox API [15]. On Android, a tool like SE Android [21] could be used. However, we leave implementing the sandbox policy itself (a slow process of evaluating the security of each system call) to future work. Projects such as Chromium have successfully developed such policies for their security goals.

The supervisor controls the setup of each of the subprocesses, and can additionally add ingress and egress filters on the recognizer module, allowing arbitrary filtering of the video data before it is retrieved by the recognizer or before the recognizer's output is sent to the application module. We implemented an example ingress filter which blurs all faces in each captured frame using OpenCV's VideoCapture class. We leave a full exploration of sensor filtering to future work.

Our implementation is lightweight and relatively simple. Our supervisor is only 228 SLOC. All of our privilege-separated Python implementations of the 17 applications in our corpus come to a total of 1,140 SLOC.

3.2 Case Study: Face Detection

Converting a computer-vision-based application for use in our architecture is a simple process. Listing 1 shows the (simplified) source code for our face detector application. The application loads a Haar cascade classifier, opens the camera, and reads video frames from the camera. For each frame, it detects all faces using the Haar cascade, draws their bounding rectangles on a blank frame, and displays the result to the user.

To convert this for use in our architecture, we break it into two pieces, roughly at the "split point" shown in Listing 1. Listing 2 shows the code for the recognizer module: it handles all steps through computing the bounding rectangles of the faces in the frame, and then simply sends this list of rectangles over its socket to the supervisor proxy. Listing 3 shows the code for the application module: it polls for objects on its socket, and then draws all bounding boxes received on a blank frame and displays the result to the user. Notably, the application module never receives images of the

```
def main():
  cascade = cv2.CascadeClassifier(cascade_file)
  capture = cv2.VideoCapture()
  capture.open(args.video)
  while True:
    retval, frame = capture.read()
    gray = cv2.cvtColor(frame,
                        cv2.COLOR_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    faces = detect(gray, cascade)
    # Split point
    height, width, depth = 720, 1280, 3
    blank = np.zeros((height, width, depth),
                     np.uint8)
    draw_rects(blank, faces, (255, 255, 255))
    cv2.imshow("face_detector", blank)
    if cv2.waitKey(5) == 27:
      break
```

Listing 1: A simplified version of our unmodified face detection application. The application reads a frame from the camera, detects faces, and draws the rectangular bounding box for each detected face.

```
def main(capture, send_socket):
  cascade = cv2.CascadeClassifier(cascade_file)
  while True:
    retval, frame = capture.read()
    gray = cv2.cvtColor(frame,
                        cv2.COLOR_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    faces = detect(gray, cascade)
    send_socket.send_pyobj(faces)
```

Listing 2: A simplified face detection recognizer module. The recognizer reads frames from the camera, detects faces, and then sends their coordinates to the supervisor proxy.

```
def main(recv_socket):
  while True:
    recv_socket.poll()
    faces = recv_socket.recv_pyobj()
    height, width, depth = 720, 1280, 3
    blank = np.zeros((height, width, depth),
                     np.uint8)
    draw_rects(blank, faces, (255, 255, 255))
    cv2.imshow("face_detector", blank)
    if cv2.waitKey(5) == 27:
      break
```

Listing 3: A simplified face detection application module. The application polls and receives bounding box objects sent by the recognizer through the proxy, and uses the face coordinates to draw their rectangular bounding boxes.

faces themselves, or anything other details of the raw video frames, only the locations of the faces in each frame.

4. EVALUATION

We evaluated our architecture using the DARKLY application corpus [13], a set of existing OpenCV applications spanning a wide range of algorithms and vision tasks. Because those applications are several years old and OpenCV's API has evolved since then, we updated their original C/C++ code to be fully compliant with the modern OpenCV C++ API. When applicable, we changed the output of these applications to fit the recognizer paradigm. Often these applications output the original video frame alongside the data they were extracting from it. We also added two other simple applications to complement the DARKLY applications. A full list of these changes and the additional applications we added to the corpus is in Appendix A.

For consistency, we ported each application to Python. Porting was often straightforward and followed line-for-line from the original application code, replicating the functionality and implementation as closely as possible. Then, we re-factored these implementations to be suitable for our architecture: we separated each into a recognizer and an application module.

For reproducible results, we created a 1–5 second long video for each application, tailored to have the objects or characteristics expected for video captured by that application. In our performance measurements, we used this video as the input to the application.

We ran each application in three different configurations: native (the original application, a monolithic C/C++ application), monolithic Python (our port to Python), and privilege-separated Python (our version split into separate recognizer and application modules). For each, we collected log data from 80 trials over the video file. All experiments were performed on a Raspberry Pi 2 Model B, which has an 900MHz quad-core ARMv7 CPU and 1GB of RAM, running Raspbian 7, and run in headless mode under the virtual framebuffer Xvfb. This device is similar to Google Glass in terms of hardware and performance.

4.1 Performance

For our performance experiments, the supervisor's token bucket had a fill rate and bucket size large enough to never drop any items. This allows us to separate the effects of the bandwidth limitation from the overhead of the architecture and proxy themselves.

We measured the performance of our privilege-separation architecture implementation in terms of:

- Frame rate: frames read and processed per second
- Startup time
- Proxy delay (latency)

A goal for many computer vision applications is real-time video processing, which means the processing of a single frame should take only 30–40 ms (corresponding to roughly 30 frames per second) [17]. Figure 3 shows the frame read rate for each application. This measures how many frames the application can read from its video input and process in a second.

We found that the privilege-separated Python versions were frequently even faster than their monolithic Python counterparts, and sometimes even faster than the native C++ versions. This speedup is a positive side effect of our privilege-separated architecture exposing more opportunities for parallelism, giving a speedup on multi-core CPUs. The recognizer can run in parallel with the application module, enabling both to run at higher frame rates. We verified this effect by re-running the application with all processes forced onto the same core (using CPU affinity to simulate a single-core CPU); for most applications, the performance gain disappears.

Even when restricted to a single core, the privilege-separated version can still be faster than the original monolithic version in some cases. This is a subtle consequence of how the applications in our corpus tend to be structured. To avoid monopolizing the CPU and allow interaction with the user interface, the original monolithic versions pause for 5 ms after each iteration of their processing loop[1]. In the privilege-separated version, the separate recognizer process can run and do useful work during this pause.

Overall, for most applications, privilege separation rarely slows down the application, and 5 out of the 17 applications saw a speedup of 10% or more, and 3 out of the 17 applications saw a speedup of over 30%. Thus, privilege separation often exposes useful opportunities for parallelization. This experience suggests that the performance does not need to be a barrier to deployment of our proposed architecture.

Figure 4 shows the average startup time for each application (the time from starting to reading the first frame). The startup time is very low across the board, in absolute terms, and our privilege-separation architecture has at most a modest effect on startup time. Our measurements do not include the overhead of initializing the Python VM and loading the initial imports.

Finally, the added latency of our architecture needs to be small, so that results received by the application are sufficiently "live". Low latency allows a much greater range of applications that rely on real-time contextual information. Table 1 shows the average delay introduced by our architecture for each application. This measures the time between the recognizer detecting a feature in the video frame and sending an object on its outgoing socket, and the application receiving and decoding that object. The worst delay is for ellipse-fitter, which is still less than 5ms. The supervisor delay does not appear to be affected by whether the application requires bi-directional communication between the recognizer and the application modules, or by the size of the objects being sent through the supervisor proxy. To give a point of comparison for these numbers: virtual reality (VR) applications have exceptionally strict latency requirements to maintain virtual registration, around 15 ms [3]. For all of our applications, our supervisor proxy adds minimal latency compared to this threshold.

4.2 Bandwidth

Our architecture enforces a bandwidth-limited channel between the recognizer and the application in order to restrict the recognizer from extracting unnecessary information from

[1]All of the DARKLY applications with processing loops call waitKey after each iteration of the loop, i.e., after processing each frame of video. This is a commonly recommended practice for OpenCV applications to allow interaction with the user interface. We retained this in all of our modified versions.

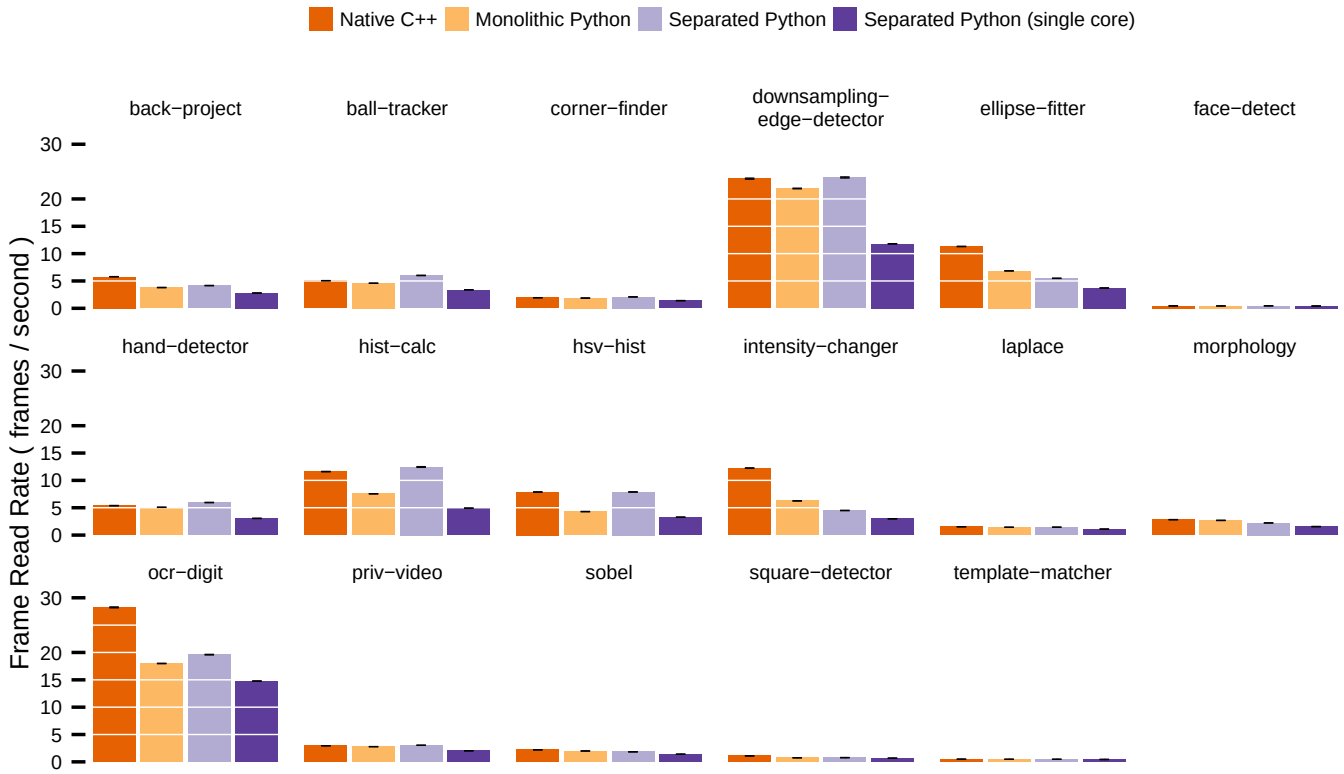


Figure 3: The frame read rate for each application, for each configuration: Native C++, monolithic Python, privilege-separated Python, and privilege-separated Python with single core CPU affinity. Higher rates are better. Each configuration was run repeatedly and averaged ($n = 80$), and the 95% confidence intervals are shown on each bar.

the raw video data. The bandwidth setting is closely related to the level of privacy and security protection provided by our architecture.

To measure the privacy levels enforced by such a channel, we considered a malicious recognizer that attempts to exfiltrate video with enough fidelity that the application could still detect faces in the output, but possibly not enough to recognize who is visible. We hypothesized that this represents a conservative lower bound for the frame rate that would be needed to capture video of a socially embarrassing or revealing moment about some human person.

We started with a test input video of a mostly stationary face in the middle of the frame. Of course, a malicious recognizer could compress the video from the camera using lossy compression and exfiltrate the compressed result, rather than raw video, thereby using less bandwidth. To reflect this, we compressed the test input video as much as possible, while ensuring that the person's face is still recognizable in the compressed video. In particular, we compressed the video using ffmpeg and libx264, trying a range of resolutions (maintaining aspect ratio) and settings, and measuring the size of the resulting compressed video. This yields much better compression ratios than compressing each frame of video separately. We applied OpenCV's facial detection algorithm to each compressed video, rejecting any compressed video where faces were no longer detectable, and kept the smallest compressed video where faces remained detectable.

Our test input video was 5.4 seconds long and 720p resolution; uncompressed, it was 6.2 MB. We found that at the optimal combination of resolution and target size (1218 x 684 pixels at 50 KB target size), we can get a 48756 byte video that still has detectable faces. At 30 frames per second, over 5.4 seconds, a malicious recognizer would need to exfiltrate 300 bytes per frame to exfiltrate this 5.4 second compressed video.

Additionally, we tested our privacy budget qualitatively by encoding a more realistic video to fit under the 300 bytes per frame budget. We used a 34 second clip from a video featuring people outdoors in natural lighting [20]. Figure 5 shows a frame from the heavily compressed video. At under 300 bytes per frame, faces and distinguishing features are heavily blurred and artifacted.

Accordingly, we set a *privacy budget* of 300 bytes per frame, and evaluated how many applications would stay within this budget. In other words, we envision that the supervisor proxy could be configured to limit the bandwidth from the recognizer to the application module to a maximum of about 300 bytes per frame, and then we analyze how many applications will continue to work without violating this budget.

To do this, we measured the bandwidth requirements of the applications we tested. Table 2 shows the average total size of objects sent for a single frame by each application, in increasing order, as well as the average sizes when we enabled zlib compression.

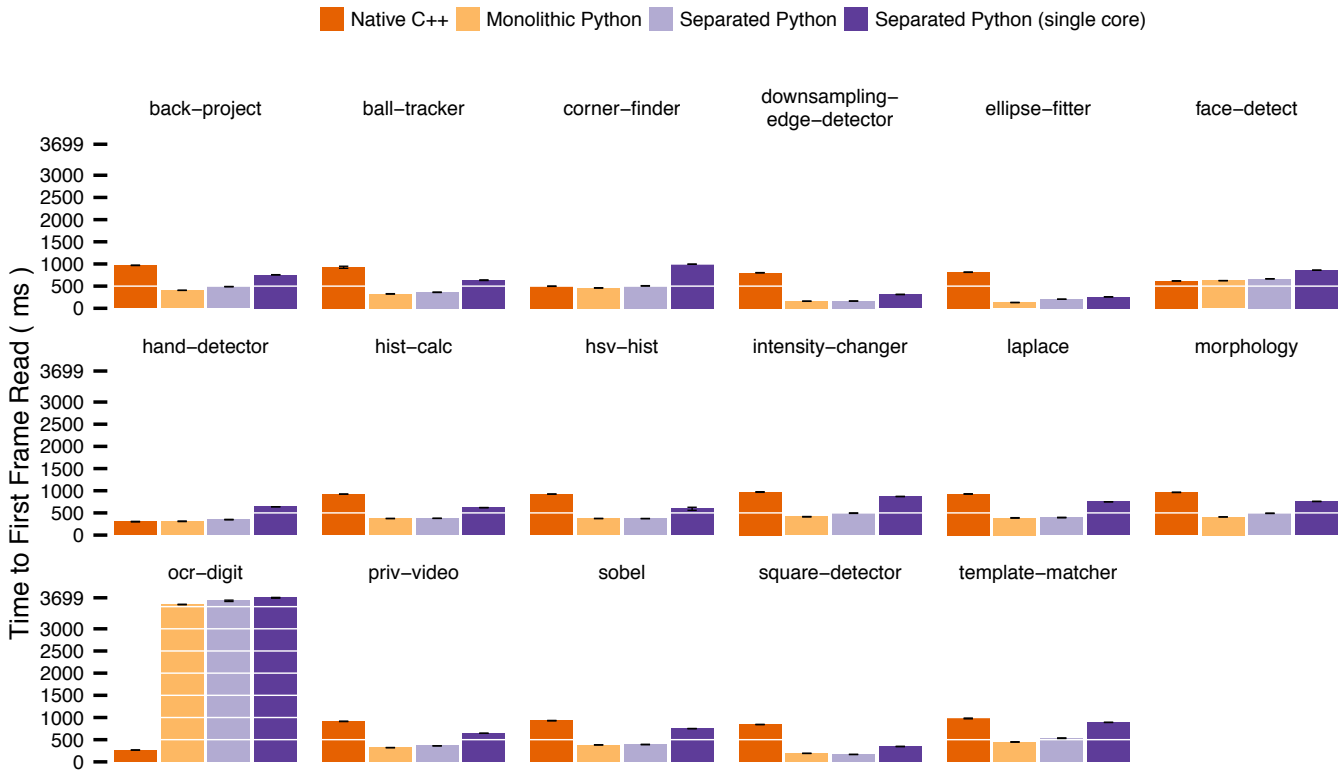


Figure 4: The startup time for each application (the mean time from starting to reading the first frame from the capture). Lower times are better. Each configuration was run repeatedly and averaged ($n = 80$), and the 95% confidence intervals are shown on each bar.

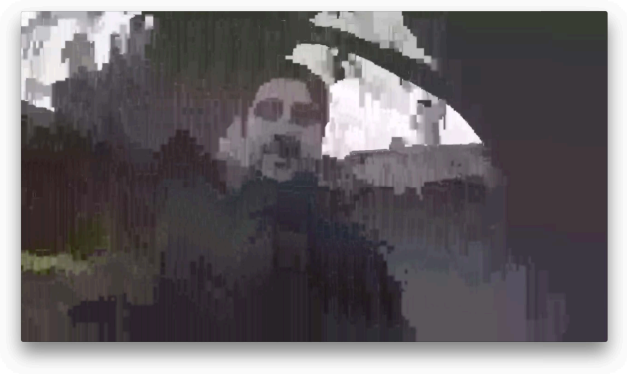


Figure 5: A frame from the realistic test video when re-encoded to fit under our 300 bytes per frame privacy budget. The original video was a 34 second clip from the full video, at 720p resolution and 27.9MB total size. Our re-encoded version was 291KB. Faces are almost completely blurred. (Frame is from the video "Jorge Aravena" by skydiveandes [20], licensed under CC BY 3.0 [5]. Compressed from original.)

We see that the applications which only extract straightforward features from the frame, such as the coordinates of a face, have very small object sizes, while those that extract features of the entire frame, such as a histogram or edges, have larger object sizes, and those that send the entire frame (or modified versions of it) in some modified form have very large object sizes. In the best case, the recognizer might only send objects very infrequently. In the worst case, a recognizer would be sending an object every frame. In the average case, this could be amortized by setting the bucket size of the supervisor proxy high enough to allow intermittent large objects.

Some applications may not function as recognizers (such as morphology) because these applications don't "recognize" anything from the video and have too high a bandwidth requirement. Instead, they could be used as a step in a more involved computer vision task in a true recognizer application. 4 of our 17 applications would work as-is even with their bandwidth limited to our privacy budget of 300 bytes/frame. 12 of our 17 applications would work if they limited their output to once every 30 frames.

We recognize that 300 bytes/frame is a highly conservative privacy budget, and a higher bandwidth limit might be relatively harmless in practice. Our test input video is based on conditions that are favorable to a malicious recognizer, likely unrealistically favorable: it exhibits little to no movement, large subjects in frame, no other background objects or activity, and clear lighting. Also, our experiment presumes the ability to do multiple encoding passes instead of a fixed

Application	Delay (ms)	±95%CI
template-matcher	0.003	±0.001
intensity-changer	0.004	±0.000
ocr-digit	0.004	±0.000
ball-tracker	0.005	±0.000
face-detect	0.007	±0.000
corner-finder	0.007	±0.000
downsampling-edge-detector	0.008	±0.000
hist-calc	0.008	±0.000
square-detector	0.029	±0.000
back-project	0.032	±0.000
hand-detector	0.039	±0.000
priv-video	0.040	±0.000
sobel	0.040	±0.000
laplace	0.112	±0.001
morphology	0.184	±0.002
hsv-hist	0.368	±0.059
ellipse-fitter	2.803	±0.057

Table 1: The average delay of sending an object through the supervisor proxy (in milliseconds). This is the time between the recognizer module marshaling and sending the object on the socket to the application module receiving the object and un-marshaling it.

constant bit rate, whereas in practice a malicious recognizer would have to compress the video in realtime and use a constant bit rate. Also, if the supervisor resets the recognizer periodically, a malicious recognizer would find it even more difficult to achieve such a compression rate. Therefore, we acknowledge that our experiment is based on conservative assumptions about the threshold of concern, and might give a unduly pessimistic view of our architecture's ability to support existing applications. We use this experiment to try to capture a conservative but objective measure of the fuzzier notion of "embarassing video": if the quality is so low that it is not even possible to recognize a face, then the potential harm of leaking the entire video is likely to be fairly low.

4.3 Developer burden

Refactoring the Python implementations for our architecture was often as simple as splitting the application into two pieces and adding three lines for socket communication (send, poll, and receive in the main loops). We presented a case study of converting the face detection application in Section 3.2. Our experience was that converting applications to our architecture was usually straightforward. The more complicated cases involved bi-directional communication between the recognizer and the application and vice-versa; these required slightly more complicated handling of the multiple sockets to avoid blocking, but ultimately each of these applications reused most of the same code for handling this.

This experience suggests that in most cases our architecture would not impose onerous burdens on application developers. Converting existing applications could be done with a modest effort, and we expect that developing new applications with this architecture in mind might be even easier.

Application	Object size (bytes)	Compressed (bytes)
ocr-digit	48.00	56.00
ball-tracker	56.00	64.00
template-matcher	62.39	72.00
face-detect	184.00	192.00
corner-finder	344.00	319.13
intensity-changer	1,200.00	365.96
hist-calc	3,320.00	2,999.18
hsv-hist	4,016.00	2,016.19
square-detector	12,726.38	6,651.10
downsampling-edge-detector	76,976.00	4,545.48
ellipse-fitter	122,124.82	45,179.60
hand-detector	921,776.00	1,388.73
priv-video	921,776.00	9,023.52
sobel	921,776.00	491,073.79
back-project	921,928.00	83,192.45
laplace	2,764,976.00	944,050.47
morphology	5,529,816.00	2,122,782.51

Table 2: The average size of objects (in bytes) sent over the supervisor proxy for each application, sorted by increasing size.

4.4 Security Analysis

For our privilege-separation architecture, we consider the following attacker goals:

- Record embarrassing video and exfiltrate it, or
- Extract confidential information (e.g., a credit card number) and exfiltrate it.

We consider two distinct threat models: one in which the application module is compromised or under attacker control, or careless with its data; and one in which the recognizer is compromised or under attacker control, or careless with its data.

In the first threat model, with a compromised application module, our architecture prevents both of the attacker goals. The attacker cannot change what features the recognizer extracts from the video, and only has access to the recognizer output. In a monolithic design, an attacker would succeed at both goals.

In the second threat model, with a compromised recognizer module, our architecture would prevent the exfiltration of embarrassing video, but would not protect against the exfiltration of confidential extracted information. The attacker could change the recognizer's code to extract the desired sensitive information, but would not be able to bypass the bandwidth restriction of the supervisor proxy. Again, in a monolithic design, an attacker would succeed at both goals.

5. DISCUSSION

Privacy. We make the distinction between two kinds of privacy breaches: leaking specific short secrets, and leaking embarassing video. Our architecture helps with the latter but not the former. For example, it does not prevent leakage of a credit card number (extracted from a credit card in the

image) or key strokes (extracted from a visible keyboard) or a house key bitting code (extracted from a picture of a house key). It does prevent leakage of a video of someone acting stupidly, or a video of someone in a compromising position (such as in a state of undress), or a family member spying on and recording someone.

Applicability. We manually reviewed the 20 applications in the DARKLY corpus. Two were no longer available, one could not be compiled, and two could not directly fit into our recognizer paradigm (a QR decoder and an application which takes two images and adds them together, outputting the result). The remaining 15 applications were readily convertible to our architecture. We believe this is a strong argument for the applicability of our architecture to existing OpenCV applications, even if they are not necessarily performing "recognition" tasks. A full list of the applications in our corpus is in Appendix A.

Recognizer state. We want to be able to limit correlation and long-term monitoring within the recognizer module. Limiting the state of the recognizer and periodically resetting it (made possible by the functional design of recognizer modules) could protect against these threats.

An ideal recognizer would have no state, or at least only initial state (e.g., loading a classifier model). However, it is often the case that a recognizer maintains some amount of state from frame to frame (such as details of several past frames in order to track foreground motion). Only two applications in our corpus maintained any state across frames; both of these would work with limited or periodically reset state.

To efficiently reset the recognizer, we propose a *hot swapping* mechanism, where a new instance of the recognizer is started before the reset, and fed the same input frames as the old instance. Then, at the reset, the recognizer-to-proxy socket is disconnected from the old instance and connected to the new instance. This way, the new instance has had time to "warm up" its state before its output is used by the application, and there is no startup cost when resetting.

Filtering. Our implementation includes a proof-of-concept option to blur all faces in each captured frame. Other possible filters include background removal (this could work particularly well for perceptual input applications, which only want to detect foreground movement) or text blurring (e.g., to prevent accidentally leaking credit card numbers or sensitive documents). These kinds of filters perform a similar function to the "noisy permissions" of Jana *et al.* [12]. Egress filtering (filtering the output of the recognizer) is much more challenging, due to the lack of semantics on the data being sent through the supervisor. DARKLY accomplished egress filtering (what they call "declassifiers") by explicitly adding filters to each requisite OpenCV function [13]. Our generality means we do not impose any semantics on the recognizer result, making egress filtering much more challenging. However, we feel that ingress filtering can accomplish many of the same goals as DARKLY's declassifiers.

Blackbox differential analysis. In our architecture, the recognizer module operates as a sort of pure function over the camera input. This could allow us to analyze a recognizer as a black box, without needing to analyze the source code. By controlling the frames inputted to a recognizer, it would be possible to determine what the behavior of the recognizer is. For example, if switching only the faces on an input frame causes a change in the recognizer output, one might be able to conclude that the recognizer is extracting facial information. This technique could be automated and used by app store reviewers, or to inform users of the information extracted.

Support for native applications. Allowing native code modules in our architecture could be useful for cases where optimizing the Python implementations is not sufficient for performance requirements. Some abstractions might be needed for marshaling data in a cross-compatible manner if the recognizer and application modules are written in different languages. To support native modules, our supervisor proxy can function as is, but after spawning the subprocesses for the recognizer and application modules, it would execute native executables instead.

6. CONCLUSION

Our privilege-separation architecture helps to secure visual recognizer applications while allowing generic and novel computation on the video input. Our evaluation finds that our architecture is practical, has a modest performance impact, requires little developer burden, and provides significant privacy and security benefits.

Acknowledgements

This work was supported by Intel through the ISTC for Secure Computing. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel.

Availability

The source code for the implementation of our architecture and all of the applications, along with the scripts we used to perform the evaluation, are available online at https://github.com/christhompson/recognizers-arch.

7. REFERENCES

[1] Layar. https://www.layar.com/, 2014.
[2] Meta augmented reality glasses. https://www.spaceglasses.com, 2014.
[3] M. Abrash. Latency – the sine qua non of AR and VR. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/.
[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security (NDSS) Symposium*, 2010.
[5] C. Commons. Creative commons attribution 3.0 unported (cc by 3.0). https://creativecommons.org/licenses/by/3.0/.
[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

Application	Modifications
back-project	Only output image and back projection
ball-tracker	Only output tracked path
corner-finder	Only output corner marks
downsampling-edge-detector	Only output reduced image of edges; use C++ API
ellipse-fitter	Only output contours image; use C++ API
hand-detector	Only output mask; use C++ API
hist-calc	Use video input, only output histrogram; use C++ API
hsv-hist	Use video input, only output histogram
intensity-changer	Only output histogram; use C++ API
laplace	Use C++ API
morphology	Only output eroded/dilated image
ocr-digit	Use video input, output only digit as text; use C++ API
priv-video	Only output foreground of frame; use C++ API
square-detector	Only output outlines; use C++ API
template-matcher	Only output template outline
security-cam	*Not used, cannot compile*
facial-features	*Not used, rebuilt as face-detect*
face-recognizer	*Not used, code no longer available*
image-adder	*Not used, does not fit recognizers paradigm*
qr-decoder	*Not used, code no longer works*

Table 3: The Darkly corpus, and our changes to each, if used.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[9] Google. Content scripts. http://developer.chrome.com/extensions/content_scripts.html.

[10] Google Inc. Google glass. http://www.google.com/glass/, 2014.

[11] itseez. OpenCV. http://opencv.org, 2014.

[12] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[13] S. Jana, A. Narayanan, and V. Shmatikov. A Scanner Darkly: Protecting user privacy from perceptual applications. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[14] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Proceedings of the 20th Network and Distributed System Security (NDSS) Symposium*, 2013.

[15] Mac Developer Library. sandbox(7). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man7/sandbox.7.html.

[16] Microsoft Coportation. XBox Kinect. http://www.xbox.com/en-US/kinect, 2014.

[17] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Realtime computer vision with OpenCV. *ACM Queue*, 10(4), April 2012.

[18] N. Raval, L. Cox, A. Srivastava, A. Machanavajjhala, and K. Lebeck. 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.

[19] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 224–238, 2012.

[20] skydiveandes. Jorge Aravana. https://vimeo.com/172654114.

[21] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible mac to android. In *Proceedings of the 20th Network and Distributed System Security (NDSS) Symposium*, 2013.

[22] R. Templeman, M. Korayem, D. Crandall, and A. Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*, 2014.

APPENDIX

A. APPLICATION CORPUS

Table 3 lists the applications we used from the Darkly corpus, and the modifications we made to them. It also lists the applications from the Darkly corpus we did not use, and our reasons for omitting them. Many of the Darkly applications required conversion to the modern OpenCV C++ API (from of the outdated C API) in order to maintain parity with our Python versions.

In addition to the Darkly applications, we added the following to our testing corpus:

- face detection (face-detect)
- sobel edge detector (sobel)