Security Architecture for Visual Recognizer Applications CS262 Fall 2013

Christopher Thompson and Rebecca Pottenger
University of California, Berkeley
cthompson@cs.berkeley.edu, rpotteng@eecs.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, applications tend to have course-grained access to sensor feeds, exposing more information than is necessary for the functionality. We propose a sandbox architecture for visual recognizer applications that strongly encourages modularization and least privilege – separating the recognizer logic and sandboxing it to restrict filesystem and network access, and limiting the bandwidth of its output. We analyzed 16 of the applications in the DARKLY corpus and show that our architecture could easily be used by all of them. We built a prototype on Mac OS X that securely sandboxes the recognizer and enforces the bandwidth limitation, and evaluated five example applications using our architecture. Our prototype incurs low overhead for each of these applications.

1. INTRODUCTION

With the increasing use and development of applications that use potentially sensitive sensor data, there is a growing need for an architecture that can protect users from applications that, while non-malicious, may not be 100% trustworthy. This is particularly true for smartphones, where the incidence of malware has been very low [9]. Far more common are apps which are accidentally over privileged [5] or misuse personal information and identifiers [4]. These applications might not intend to leak sensitive data or perform malicious actions, but might end up doing so inadvertently.

Recognizers, as introduced by Jana *et al*. [6], are a privilege separation architecture for augmented reality applications developed to address these problems. A recognizer contains a single task to be performed on sensor data: a recognizer "recognizes" and outputs certain features the application wants access to. Jana *et al*.'s design focused on computer vision tasks, and looked at a limited variety of Kinect applications. They argued for a fixed set of system provided recognizers; that is to say, eight computer vision tasks (e.g., hand tracking, skeleton detection) they found that met all of the applications' needs. However, their analysis focused on games using Kinect, and we argue that novel applications could easily require new recognizers, or customized version of existing recognizers, that their system did not anticipate.

We extended the recognizer model to allow for novel recognizers that the system may not anticipate, by developing a general security architecture that requires applications to separate out computer vision tasks into a separate "recognizer" module. Our architecture is intended for applications that operate on rich sensor data, with the aim to avoid non-malicious yet potentially harmful mistakes by the developer. This system strongly encourages privilege separation and least privilege of the components; additionally, this separation allows the system to strongly sandbox the recognizer code, preventing actions such as network or filesystem access. The sandboxing enforces a limited output bandwidth, which would cause most developers to place the code that needs access to the raw sensor streams inside the sandbox, and therefore only extract the minimal features required for their general functionality, while still allowing general computation on sensor streams. With this separation of the recognizer code, more policy, fine-grained permissions, and filtering can be applied to the sandbox. We evaluate our architecture on applicability and performance.

2. BACKGROUND

In this section, we discuss the background research relevant to our work. We describe a few papers that overview the main work done to build secure systems for untrusted third-party applications that take in potentially sensitive sensor data as input.

Traditionally, a third-party application's access to sensor data has been dependent on user permissions. For example, iOS will display a prompt requesting the user to decide whether to give a particular application access to the sensor data upon the application's first request (and then never ask again), and Android will display a prior-to-installation manifest outlining the sensor usage that application wants and will only permit installation if the user agrees to this sensor usage. These types of systems 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, a new style of permission granting, using access control gadgets, was developed by Roesner *et al*. [11]. 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 permission granting is built into the actual 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 has the potential to provide a great deal more information to the application than the application explicitly needs. An alternative to this type of system is discussed in TaintDroid [3], where 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. Yet another alternative is differential privacy [2], where adding a specifically calibrated amount of noise to the data

of all users guarantees that nothing specific (and therefore nothing sensitive) can be learned about any one individual user.

DARKLY [7] combined many of these techniques – access control, algorithmic transformation, and user audit – to provide privacy protection and privilege separation by splitting their system 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 use this same application corpus as a way to evaluate the applicability of our system.

Jana *et al.* [6] introduced the abstraction of the "recognizer". The recognizer takes the raw sensor data as input, and produces higher-level objects (e.g. a face, or hand tracking) as output. The recognizer allows for a fine-grained permission system, as the applications are only allowed to request higher-level recognizer object data. Jana *et al.*argued for a small fixed set of system provided recognizers – common computer vision tasks they found that met all of their applications' needs (e.g., hand tracking, skeleton detection, etc.). 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.

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 – 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 [1].

Compared to previous security mechanisms for visual recognizer applications [6], we allow fully general programming of the recognizer – developers can come up with novel recognizers or customized versions of their existing ones, instead of having to rely on a fixed set of system provided recognizers. We focus on a much more generalizable app framework than either DARKLY or system-provided recognizers. Our architecture must provide secure access to the live camera, while allowing easy use by apps that must interact with other libraries and services.

Our design is summarized in Figure 1. The recognizer sends data 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 forwarded the entire video stream to the application module, maintaining separation of privilege for the entire application. Second, it can limit the amount of information from the camera than could be exfiltrated by the application.

3.1 Threat Model

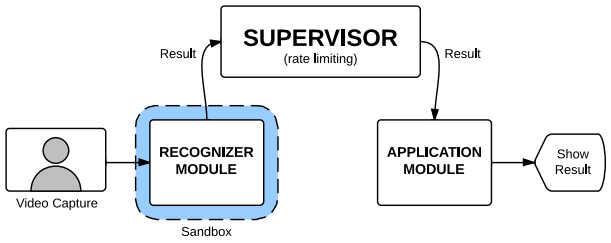

Figure 1: Our sandbox design. The recognizer module has access to the camera (potentially encapsulated to filter the frames before the recognizer gets them), but is securely sandboxed to prevent network and filesystem access. The application module receives the results of the recognizer through the supervisor proxy, which can limit the rate at which the recognizer can send data.

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). An application might unintentionally expose raw video data to cloud services or third parties (potentially through advertising code). A poorly implemented application might even later be compromised and used to attempt to over-access the camera. Our sandbox architecture protects against all of these threats.

3.2 Implementation

Our supervisor, written in Python[1], sets up the environment, spawns a child process for the recognizer and the application modules, and initializes the sandbox in the recognizer process before passing off control to the recognizer module.

The supervisor also acts as a proxy for all data sent between the recognizer and the application. The sockets are named IPC sockets created using zeromq[2] by the supervisor 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 can cause dropped frames (see Section 5.2 for more on the effects of limiting the bandwidth), 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 unmarshaling.

The recognizer is sandboxed using the OS X sandbox API [10], which is the same mechanism used by Chromium for their OS X sandbox [12]. It is similar to Linux's seccomp-bpf, and is configurable with policy files written in a Scheme-based DSL. Allowed APIs can be "warmed up" by accessing them in the process before the sandbox is initialized. Our implementation uses this API to restrict the recognizer module to read-only filesystem access, no network access, and only access to the `cv2` and `numpy` Python modules.

[1]Python 2.7.6

[2]**`http://zeromq.org`**, pyzmq version 14.0.1

```
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.cv.CV_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    faces = detect(gray, cascade)
    # Split point
    height, width = 720, 1280
    blank = np.ones((height, width, 3),
                    np.uint8)
    blank[:, 0:width] = (255, 255, 255)
    draw_rects(blank, faces, (0, 255, 0))
    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 capture device, and draws the rectangular bounding box for each detected face.

The supervisor controls the setup of each of the subprocesses, allowing it to add ingress and egress filters on the recognizer module. We implemented an ingress filter to blur all faces in each captured frame as a simple wrapper around OpenCV's `VideoCapture` class, overriding the `read()` function. Egress filters could be added into the supervisor proxy.

We chose Python and zeromq on OS X for ease of implementation, and it allows for easy portability across desktop platforms (simply by changing the sandbox implementation). Of particular interest is that OpenCV has Java and Objective-C bindings – our design could be easily ported for use in 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 (only our ingress filter).

Our implementation is lightweight and relatively simple. Our supervisor is only 179 SLOC, our sandbox API wrapper is 33 SLOC, our face blurring filter is 50 SLOC, and all of our example apps come to a total of 532 SLOC.

3.3 Case Study: Face Detection

Converting a visual recognizer application for use in our sandbox is a simple process. Listing 1 shows the (simplified) source code for our face detector application. It loads a Haar cascade classifier, opens the camera, and then for each frame read from the camera detects all faces, 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 the list of face rectangles over it's socket to the supervisor proxy. Listing 3 shows the code for the application module: it simply polls for objects on its socket, and then draws all faces received on a blank frame and displays the result to the user.

```
def main(capture, send_socket):
  cascade = cv2.CascadeClassifier(
      cascade_file)
  while True:
    retval, frame = capture.read()
    gray = cv2.cvtColor(frame,
        cv2.cv.CV_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 capture device, 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 = 720, 1280
    blank = np.ones((height, width, 3),
                    np.uint8)
    blank[:, 0:width] = (255, 255, 255)
    draw_rects(blank, faces, (0, 255, 0))
    cv2.imshow("face_detector", blank)
    if cv2.waitKey(5) == 27:
      break
```

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

This conversion required identifying where the recognition result is finished, and adding the sending and receiving code, which is made simple with the explicitly passed in sockets to the recognizer and application modules.

4. EVALUATION

We evaluated our implementation on five example applications in Python using the popular OpenCV computer vision library: background detection, edge detection, face detection, face recognition, and identity (which simply displays the captured frame). The first four are common recognizer tasks, and the last allowed us to stress the overhead of our supervisor proxy.

We refactored each of our example applications to fit our sandbox architecture: separating the video capture and vision processing into the recognizer module, and then sending the results over the supervisor proxy to the application for display. Refactoring our example applications was as simple as splitting the application into two pieces and adding three lines for socket communication (send, poll, and receive in the main loops). For each application, the total SLOC of the split modules is actually less than for the unmodified application (generally due to startup and import code being handled by the supervisor). We present a case study of converting the face detection application in Section 3.3.

In order to have reproducible results, we created a one second long video and used it as input to the OpenCV VideoCapture. This 1280x720 video shows a well-lit face staying mostly still in the center of the frame, which allows us to test all features of our example applications.

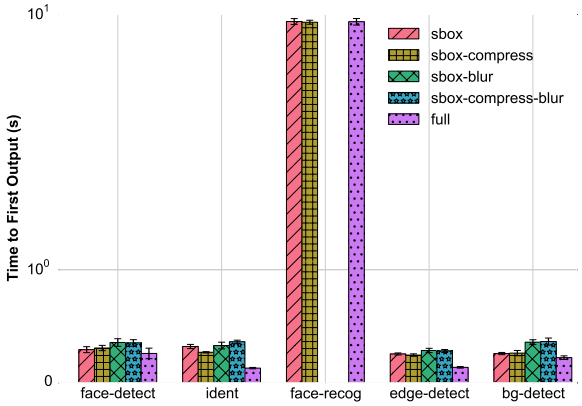

Figure 2: Mean startup time for each app and variation with standard deviations. Note that face-recog never outputs anything with face blurring enabled (the recognizer sends each face recognized individually), while face-detect sends the empty list when no faces are detected.

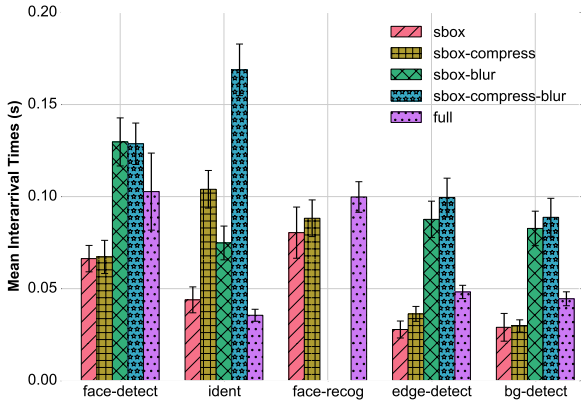

Figure 3: Inter-arrival times. 30 frames per second (a standard camera frame-rate) corresponds to an inter-arrival time of 0.033s.

We ran each application in five different configurations: unmodified, and four variations of the sandbox (with and without compression over the socket, and with and without our face blurring ingress filter enabled). For each, we collected log data from ten runs over the video file. All experiments were performed on an early 2012 Macbook Pro with a 2.2 GHz Intel Core i7 and 16GB of RAM.

For our experiments, unless otherwise noted, the supervisor's token bucket has 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 sandbox and proxy themselves. We measured the overhead of our sandbox implementation in terms of startup time, inter-arrival time (processing rate), and proxy delay (latency). We also measured the effects of limiting the bandwidth through the proxy for varying token bucket configurations, and the impact of ingress filtering.

4.1 Startup

Figure 2 shows the startup time for each application – the time until the first application output. For some applications, our sandbox incurs a significant but small startup overhead. The highest relative startup overhead was for "ident", where sbox-compress-blur took 2.8 times longer on average than the unmodified application (0.36s, $\sigma = 0.014$ vs. 0.13s, $\sigma = 0.006$). Note that for face recognition, the application with the largest startup time, sandboxing only incurred an average additional 0.02s startup time ($\sigma = 0.31$)

4.2 Inter-arrival Times

Figure 3 shows the mean time between application outputs for each application. A goal for many computer vision applications is realtime video processing, which means the processing of a single frame should take only 30-40 ms [8]. Our results for some configurations come close to this goal (simple frame filtering such as Canny edge detection and background segmentation), while model-based tasks have longer processing times.

4.3 Proxy Delay

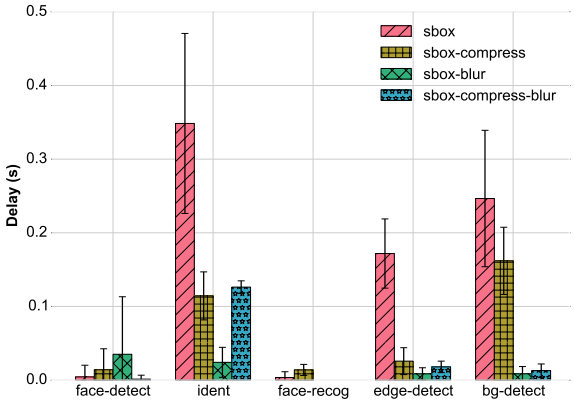

Figure 4: Mean time between the recognizer marshaling and sending an object and the application receiving and unmarshaling it.

The added latency of our sandbox 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. Figure 4 shows the mean delay incurred by sending an object over the supervisor proxy.

For applications that output frames or modified frames (ident, edge-detect, bg-detect), compression greatly reduces the data sent across the proxy, decreasing the latency. For applications that output small features, compression has no significant effect.

5. DISCUSSION

5.1 Applicability

We manually analyzed 16 of the applications from the DARKLY paper, to determine whether the applications could work with our sandbox. Table 1 summarizes our analysis. For the applications that took image files as input, we analyzed them as if they took individual frames from a video capture.

Application	Inputs	Outputs	State	Filterable?
OCR for hand-drawn digits	mouse	classification	N/A	BR
Ball tracker	video	position, path, frame	previous position	BR
QR Decoder	video	decodings, outlines, frame	N/A	BR
PrivVideo	video	foreground in black and white	previous frame	BW, BR
Histogram (RGB)	image	histogram	N/A	BR
Histogram (H-S)	image	image, histogram	N/A	BR
Square detector	image(s), parameters	outlines on image	N/A	BR
Morphological transformer	image, parameters	eroded/dilated image	N/A	BR
Intensity/contrast changer	image, parameters	greyscale image, histogram	N/A	BR
Pyramidal downsampler + Canny edge detector	image	image, reduced edges image	N/A	BW, BR
Image adder	2 images	overlap	N/A	BR
H-S histogram back-projector	image	image, histogram, histogram mapped onto image	N/A	BR
Template matcher	image, template	image with template outlined	N/A	BR
Corner finder	image	image with corners marked, number of corners detected	N/A	BR
Laplace edge detector	video	edges image	N/A	BR
Ellipse fitter	image, level	image in grayscale, contours image	N/A	BR

Table 1: Survey of inputs, outputs, and state requirements for the DARKLY corpus, and what possible filters could be used (BW = bandwidth restriction; BR = background removal).

Figure 5: Original 720p image of a face, totaling 254 kilobytes.

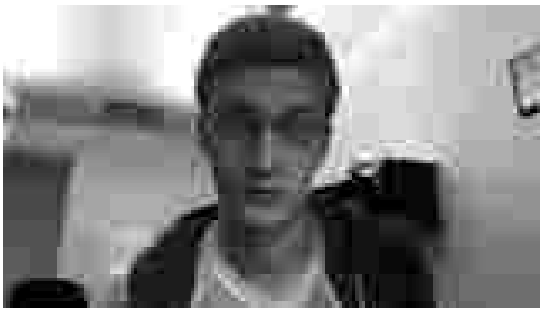

Figure 6: The same face saved as JPEG with quality of 11 and height of 90 pixels. The face is still recognizable at this level. This image only takes up 993 bytes.

We found that all of the applications could work within our framework. 15 of the applications were written in such a way that they could easily be split into separate recognizer and application modules. The last application (QR Decoder) was written in a way that would make it more difficult to split. Of the four applications that took video frames as input, none of them had a small size output (i.e., <1KB per video frame). Two of these applications would have small output if they did not output the entire frame. Of the 12 other applications, all produced small output. Four of the applications have communication from the GUI to the recognizer; our sandbox does not currently support this, but it could be added to our implementation. Only two applications maintained state; both of these would work with limited or resetting state.

5.2 Bandwidth Limits

How does one choose a bandwidth limit? How much bandwidth does an application need to exfiltrate a video stream (possibly highly compressed and reduced in resolution)? We performed a rudimentary experiment to test how much an image of a face can be compressed and resized before the face is no longer detected, or no longer recognized with the same label. For both, the smallest we could reduce the image was to 993 bytes, at a height of 90 pixels and a JPEG quality of 11. As the test image was of a face taking up most of the frame, this is somewhat a worst-case for our system; it also does not account for other features that could be lost in lossy-compression (e.g., numbers might disappear, background might be indistinguishable). But it seems a lower bound of 900 bytes per frame is needed to usefully exfiltrate a complete video stream. Figure 5 shows the original image, and Figure 6 shows the compressed

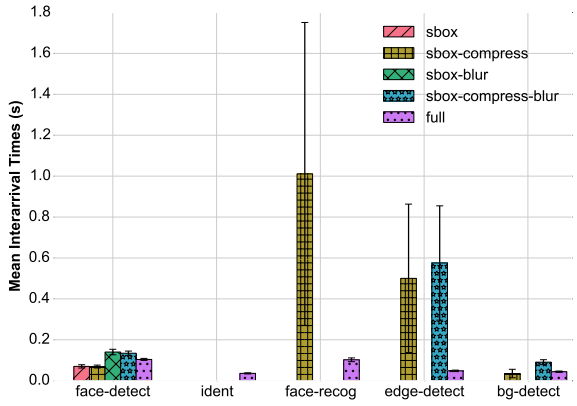

Figure 7: Inter-arrival times under limited bandwidth (token fill rate of 27000, bucket size of 54000). 30 frames per second (a standard camera frame-rate) corresponds to an inter-arrival time of 0.033s.

and resized image.

Imposing a 900 bytes per frame limit, at a capture rate of 30 frames per second, means a token fill rate of 27000 tokens per second. We repeated our experiments from Section 4.2 with a fill rate of 27000 and a bucket size of 54000 – the results are shown in Figure 7. Note that no output is ever produced for ident when in the sandbox – it never has enough tokens to send a frame to the application. The output from face-detect is small enough to pass through the bandwidth limitation even when uncompressed, and achieves roughly the same rate as when given unlimited bandwidth. The other applications function when they use compression.

Setting the bandwidth limit higher can still be useful – resizing and JPEG compressing each frame is much more effort than simply building the recognizer to properly use our sandbox. Additionally, ingress filters can provide stronger privacy protections while allowing higher bandwidths.

5.3 Filtering

Our implementation includes an 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).

Egress filtering – that is, 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 called "declassifiers") by explicitly adding filters to each requisite OpenCV function. Our generality means we do not impose any semantics on the recognizer result, making egress filtering much more challenging. However, ingress filtering can accomplish the same goals as DARKLY's declassifiers.

5.4 Future Work

While our initial results have shown that our sandbox architecture is viable, there are many further applications and evaluations we would like to explore.

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 causing a change in the recognizer output, one could 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.

DARKLY corpus: While we have manually surveyed the DARKLY corpus and determined that all of the apps (or some variation for use on camera input) could be easily ported to our architecture, we want to implement and evaluate all of them in order to have a broader evaluation of the performance and applicability of our architecture.

Compare to C++ versions: To fully understand the overhead of our architecture, we need to build native versions of each app and measure their performance. This will allow us to ensure that our percentage overhead numbers are not being washed out by the absolute overhead of using Python.

Support for native apps: While this would be a simple modification to our implementation (the supervisor can still be in Python, but after spawning subprocesses it execs the native executables instead), it would demonstrate the flexibility of our architecture, and allow for higher performance when needed.

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.

Bidirectional communication: An application might want to communicate back to the recognizer (e.g., to trigger the recognizer on some GUI, timer, or other non-continuous event). Visual recognizer applications are somewhat inverted from browser extensions – the application code is the potentially exploited module, and the recognizer code is the module with the privileges we are trying to protect. There are privacy and security implications of allowing the application to communicate to the recognizer that would need to be investigated. The actual implementation would be quite simple: an extra pair of sockets flowing in the opposite direction, routed through the proxy.

6. REFERENCES

[1] Content scripts. `http://developer.chrome.com/extensions/content_scripts.html`.

[2] C. Dwork. The differential privacy frontier. In *Theory of cryptography*, pages 496–502. Springer, 2009.

[3] 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 '10, 2010.

[4] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium*, Sec '11, 2011.

[5] 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 '11, 2011.

[6] 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*, Sec '13, 2013.

[7] 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*, S&P '13, 2013.

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

[9] 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 & Distributed System Security Symposium*, NDSS '13, 2013.

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

[11] 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 *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 224–238. IEEE, 2012.

[12] The Chromium Projects. OSX sandboxing design. **http://www.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design**.