A Scanner Darkly: Protecting User Privacy From Perceptual Applications

Suman Jana* Arvind Narayanan† Vitaly Shmatikov*
*The University of Texas at Austin
†Princeton University

***Abstract*—Perceptual, "context-aware" applications that observe their environment and interact with users via cameras and other sensors are becoming ubiquitous on personal computers, mobile phones, gaming platforms, household robots, and augmented-reality devices. This raises new privacy risks.**

We describe the design and implementation of DARKLY, a practical privacy protection system for the increasingly common scenario where an untrusted, third-party perceptual application is running on a trusted device. DARKLY is integrated with OpenCV, a popular computer vision library used by such applications to access visual inputs. It deploys multiple privacy protection mechanisms, including access control, algorithmic privacy transforms, and user audit.

We evaluate DARKLY on 20 perceptual applications that perform diverse tasks such as image recognition, object tracking, security surveillance, and face detection. These applications run on DARKLY unmodified or with very few modifications and minimal performance overheads vs. native OpenCV. In most cases, privacy enforcement does not reduce the applications' functionality or accuracy. For the rest, we quantify the tradeoff between privacy and utility and demonstrate that utility remains acceptable even with strong privacy protection.

I. INTRODUCTION

Modern software programs increasingly include *perceptual* functionality that takes advantage of high-resolution cameras and other sensors to observe their users and physical environment. Perceptual software includes "natural user interface" systems that interact with users via gestures and sounds, image recognition applications such as Google Goggles, security software such as motion detectors and face recognizers, augmented reality applications, "ambient computing" frameworks, a variety of video-chat and telepresence programs, and other context-aware software.

Hardware platforms for perceptual applications include mobile phones, programmable robotic pets and household robots (e.g., iRobot Create platform), gaming devices (e.g., Kinect), augmented reality displays (e.g., Google Glass), and conventional computers equipped with webcams. Many platforms provide app stores—for example robotappstore.com ("your robots are always up-to-date with the coolest apps")—enabling consumers to download and execute thousands of third-party perceptual applications.

The growing availability and popularity of potentially untrusted perceptual applications capable of scanning their surroundings at fine level of detail—and, in the case of programmable robots, even moving around—raises interesting privacy issues for their users. Many people are already uncomfortable with law enforcement agencies conducting large-scale face recognition [2, 17]. Perceptual applications running in one's home or a public area may conduct unauthorized surveillance, intentionally or unintentionally overcollect information (e.g., keep track of other people present in a room), and capture sensitive data such as credit card numbers, license plates, contents of computer monitors, etc. that accidentally end up in their field of vision.

General-purpose, data-agnostic privacy technologies such as access control and privacy-preserving statistical analysis are fairly blunt tools. Instead, we develop a *domain-specific* solution, informed by the structure of perceptual applications and the computations they perform on their inputs, and capable of applying protection at the right level of abstraction.

Our system, DARKLY, is a privacy protection layer for untrusted perceptual applications operating on trusted devices. Such applications typically access input data from the device's perceptual sensors via special-purpose software libraries. DARKLY is integrated with OpenCV, a popular computer vision library which is available on Windows, Linux, MacOS, iOS, and Android and supports a diverse array of input sensors including webcams, Kinects, and smart cameras. OpenCV is the default vision library of the Robot Operating System (ROS); our prototype of DARKLY has been evaluated on a Segway RMP-50 robot running ROS Fuerte. DARKLY is language-agnostic and can work with OpenCV programs writen in C, C++, or Python. The architecture of DARKLY is not specific to OpenCV and can potentially be adapted to another perceptual software library with a sufficiently rich API.

We evaluate DARKLY on 20 existing OpenCV applications chosen for the diversity of their features and perceptual tasks they perform, including security surveillance with motion detection, handwriting recognition, object tracking, shape detection, face recognition, background-scenery removal from video chat, and others.

18 applications run on DARKLY unmodified, while 2 required minor modifications. The functionality and accuracy of most applications are not degraded even with maximum privacy protection. In all cases, performance with DARKLY is close to performance on "native" OpenCV.

DOI 10.1109/SP.2013.31

II. Threat model and design of Darkly

We focus on the scenario where the device, its operating system, and the hardware of its perceptual sensors are trusted, but the device is executing an untrusted third-party application. The application can be arbitrarily malicious, but it runs with user-level privileges and can only access the system, including perceptual sensors, through a trusted API such as the OpenCV computer vision library.

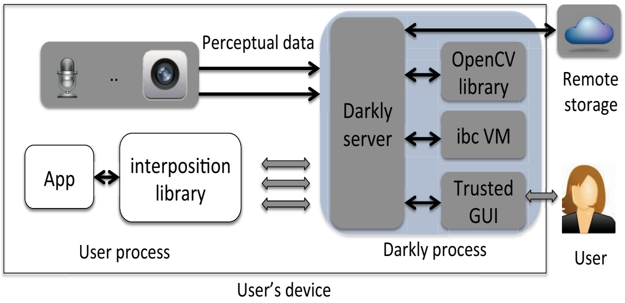


Figure 1. System architecture of Darkly.

The system model of Darkly is shown in Fig. 1 with the trusted components shaded. Darkly itself consists of two parts, a trusted local server and an untrusted client library. We leverage standard user-based isolation provided by the OS: the Darkly server is a privileged process with direct access to the perceptual sensors, while applications run as unpriviliged processes that can only access the sensors through Darkly. Furthermore, we assume that no side-channel information about Darkly operation (e.g., screenshots of its console) can be obtained via system calls. The untrusted Darkly client library runs as part of each application process and communicates with the Darkly server. This is merely a utility for helping applications access the perceptual API and the system remains secure even if a malicious application modifies this library.

A major challenge in this design is figuring out which parts of the input should be revealed to the application and in what form, while protecting "privacy" in some fashion. Visual data in particular are extremely rich and diverse, making it difficult to isolate and identify individual objects. Existing methods for automated image segmentation are too computationally expensive to be applied in real time and suffer from high false positives and false negatives.

Darkly applies multiple layers of privacy protection to solve the problem: access control, algorithmic transformation, and user audit. First, it replaces raw perceptual inputs with *opaque references*. Opaque references cannot be dereferenced by an application, but can be passed to and from trusted library functions which thus operate on true perceptual data without loss of fidelity. This allows applications to operate on perceptual inputs without directly accessing them. This approach is so natural that privacy protection is completely transparent to many existing applications: they work on Darkly without any modifications to their code and without any loss of accuracy or functionality.

Second, some applications such as security cameras and object trackers require access to certain high-level features of the perceptual inputs. To support such applications, Darkly substitutes the corresponding library API with *declassifier functions* that apply appropriate feature- or object-specific (but application-independent!) privacy transforms before returning the data to the application. Example of transforms include sketching (a combination of low-pass filtering and contour detection) and generalization (mapping the object to a generic representative from a predefined dictionary).

To help balance utility and privacy, the results of applying a privacy transform are shown to the user in the Darkly console window. The user can control the level of transformation via a dial and immediately see the results. In our experience, most applications do not need declassifiers, in which case Darkly protects privacy without any loss of accuracy and the Darkly console is not used. For those of our benchmark applications that use declassifiers, we quantitatively evaluate the degradation in their functionality depending on the amount of transformation.

Darkly provides built-in trusted services, including a trusted GUI—which enables a perceptual application to show the result of computation to the user without accessing it directly—and trusted storage. For example, after the security camera detects motion, it can store the actual images in the user's Google Drive without "seeing" them.

A few applications, such as eigenface-based face recognizers, need to operate directly on perceptual inputs. Darkly provides a domain-specific `ibc` language based on GNU `bc`. Isolating domain-specific programs is much easier than isolating arbitrary code. Untrusted `ibc` programs are executed on the raw inputs, but have no access to the network, system calls, or even system time. Furthermore, Darkly only allows each invocation to return a single 32-bit value to the application. We show that legitimate computations can be ported to `ibc` with little difficulty.

```
...
// Grab a frame from camera
img=cvQueryFrame(..);
// Process the image to filter out unrelated stuff
...
// Extract a binary image based on the ball's
    color
cvInRangeS(img, ...);
...
// Process the image to filter out unrelated stuff
...
// Compute the moment
cvMoments(...);
// Compute ball's coordinates using moment
...
// Move robot towards the calculated coordinates
...
```

Listing 1. Outline of the ball-tracking robot application.

To illustrate how DARKLY works on a concrete example, Listing II shows a simplified ball-tracking application for a robotic dog. The code on the light gray background does not need direct access to image contents and can operate on opaque references. The code on the dark gray background invokes a DARKLY declassifier, which applies a suitable privacy transform to the output of the *cvMoments* OpenCV function. The rest of the code operates on this transformed data. DARKLY thus ensures that the application "sees" only the position of the ball. The accuracy of this position depends on the privacy transform and can be adjusted by the user via the privacy dial.

III. PRIVACY RISKS OF PERCEPTUAL APPLICATIONS

> What does a scanner see? Into the head? Down into the heart? Does it see into me, into us? Clearly or darkly?
>
> *A Scanner Darkly* (2006)

Perceptual applications present unique privacy risks. For example, a security-cam application, intended to detect motion in a room and raise an alarm, can leak collected video feeds. A shape detector can read credit card numbers, text on drug labels and computer screens, etc. An object or gesture tracker—for example, a robot dog programmed to follow hand signals and catch thrown balls—can be turned into a roving spy camera. A face detector, which hibernates the computer when nobody is in front of it, or a face recognizer, designed to identify its owner, can surreptitiously gather information about people in the room. A QR code scanner, in addition to decoding bar codes, can record information about its surroundings. App stores may have policing mechanisms to remove truly malicious applications, but these mechanisms tend to be ineffective against applications that collect privacy-sensitive information about their users.

Overcollection and aggregation. The privacy risks of perceptual applications fall into several hierarchical categories. The first is overcollection of raw visual data and the closely related issue of aggregation. The problem of aggregation is similar to that of public surveillance: a single photograph of a subject in a public place might make that individual uncomfortable, but it is the accumulation of these across time and space that is truly worrying. Even ignoring specific inferential privacy breaches made possible by this accumulation, aggregation itself may inherently be considered a privacy violation. For example, Ryan Calo argues that "One of the well-documented effects of interfaces and devices that emulate people is the sensation of being observed and evaluated. Their presence can alter our attitude, behavior, and physiological state. Widespread adoption of such technology may accordingly lessen opportunities for solitude and chill curiosity and self-development." [4]

Many applications in DARKLY work exclusively on opaque references (Section VI-B), in which case the application gets no information and the aggregation risk does not arise. For applications that do access some objects and features of the image, we address aggregation risks with the DARKLY console (Section VIII). The DARKLY console is an auxiliary protection mechanism that visually shows the outputs of privacy transforms to the user, who has the option to adjust the privacy dial, shut down the application, or simply change his or her behavior. A small amount of leakage may happen before the user has time to notice and react to the application's behavior, but we see this as categorically different from the problem of aggregation. The DARKLY console is rougly analogous to the well-established privacy indicators in smartphones that appear when location and other sensory channels are accessed by applications.

Inference. The first category of inference-based privacy risks is specific, sensitive pieces of information—anything from a credit card number to objects in a room to a person's identity—that are leaked by individual frames.

DARKLY addresses such threats by being *domain*- and *data*-dependent, unlike most privacy technologies. Privacy transforms (see Section VII), specifically sketching, minimize leakage at a frame-by-frame level by interposing on calls that return specific features of individual images (see examples in Figs. 2 and 3). Privacy protection is thus specific to the domain and perceptual modality in question, and some privacy decisions are made by actually examining the perceptual inputs. In contrast to basic access control, this domain-specific design sacrifices the simplicity of implementation and reasoning. In exchange, we gain the ability to provide the far more nuanced privacy properties that users intuitively expect from perceptual applications.

The last category in the hierarchy of privacy risks is semantic inference. For example, even a sketch may allow inference of potentially sensitive gestures, movements, proximity of faces, bodies, etc. It is unlikely these risks can be mitigated completely except for specific categories of applications, mainly those that can function solely with opaque references or require only numerical features such as histograms where techniques like differential privacy [9, 10] may apply. Unless the transformed data released to the application is sufficiently simple to reason about analytically, the semantic inference risk will exist, especially due to the continual nature of perceptual observation.

That said, a machine-learning-based, data-dependent approach to privacy transforms offers some hope. For example, in Section VII-B, we describe how to use facial identification technology to transform a face into a privacy-preserving "canonical representation." The key idea here is to take a technology that leads to the inference risk, namely facial recognition, and turns it on its head for privacy protection. It is plausible that this paradigm can be extended to handle other types of inference, and as more complex inference techniques are developed, privacy transforms will co-evolve to address them. This is left to future work.

IV. Structure of perceptual applications

Darkly is based on the observation that *most legitimate applications do not need unrestricted access to raw perceptual inputs*. This is reflected in their design. For example, most existing OpenCV applications do not access raw images (see Section IX) because implementing complex computer vision algorithms is difficult even for experienced developers. Fortunately, the OpenCV API is at the right level of abstraction: it provides domain-specific functions for common image-processing tasks that applications use as building blocks. This enables applications to focus on specific objects or features, leaving low-level image analysis to OpenCV functions and combining them in various ways. Darkly ensures that these functions return the information that applications need to function—but no more!

Perceptual applications can be classified into three general categories: (1) those that do not access the perceptual inputs apart from invoking standard library functions; (2) those that access specific, library-provided features of the inputs; and (3) those that must execute their own code on raw inputs. For applications in the first category, Darkly completely blocks access to the raw data. For the second category, Darkly provides declassifier functions that apply privacy transforms to the features before releasing them to the application. For the third category, Darkly isolates untrusted code to limit the leakage of sensitive information.

For example, a security camera only needs to detect changes in the scene and invoke a trusted service to store the image (and maybe raise an alarm). This requires the approximate contours of objects, but not their raw pixels. Trackers need objects' moments to compute trajectories, but not objects themselves. A QR scanner works correctly with only a thresholded binary representation of the image, etc.

Darkly is designed to support more sophisticated functionalities, too. For example, applications dealing with human faces can be classified into "detectors" and "recognizers." Face detectors are useful for non-individualized tasks such as emotion detection or face tracking—for example, a robotic pet might continually turn to face the user—and need to know only whether there is a rectangle containing a face in their field of vision. To support such applications, Darkly provides a privacy transform that returns a generic representation of the actual face.

Face recognizers, on the other hand, must identify specific faces, e.g., for visual authentication. Even in this case, a recognizer may run an algorithm comparing faces in the image with a predefined face but only ask for a single-bit answer (match or no match). To support such applications, Darkly allows execution of arbitrary image analysis code, but rigorously controls the information it can export.

V. Design principles of Darkly

Block direct access to perceptual inputs. Darkly interposes on all accesses by applications to cameras and other perceptual sensors. As shown in Fig. 1, this privacy protection layer is implemented as a Darkly server that runs as a privileged "user" on the same device as the applications; only this user can access the sensors. Applications interact with the Darkly server via inter-process sockets (UNIX domain sockets) and standard OS user isolation mechanisms prevent them from accessing the state of Darkly.

The key concept in Darkly is *opaque reference*. Opaque references are handles to image data and low-level representations returned by OpenCV functions. An application cannot dereference them, but can pass them to other OpenCV functions, which internally operate on unmodified data without any loss of fidelity. Applications can thus perform sophisticated perceptual tasks by "chaining together" multiple OpenCV functions. In Section IX, we show that many existing applications produce exactly the same output when executed on Darkly vs. unmodified OpenCV.

A similar architectural approach is used by PINQ [18], a system for privacy-preserving data analysis. PINQ provides an API for basic data-analysis queries such as sums and counts. Untrusted applications receive opaque handles to the raw data (`PINQueryable` objects) which they cannot dereference, but can pass to and from trusted API functions thus constructing complex queries.

Darkly also provides *trusted services* which an application can use to "obliviously" export data from the system, if needed. For example, after a security-camera application detects motion in the room, it can use a trusted remote-storage service to store the captured image in the user's Google Drive—without accessing its pixels!

Support unmodified applications, whenever possible. Darkly is language-independent and works equally well with OpenCV applications written in C, C++, or Python. It changes neither the API of the existing OpenCV functions, nor OpenCV's types and data structures. Instead, opaque references replace pointers to raw pixels in the meta-data of OpenCV objects. Darkly is thus completely transparent to applications that do not access raw image data, which are the majority of the existing OpenCV applications (Section IX).

Use multiple layers of privacy protection. Applications that do not access raw inputs assemble their functionality by passing opaque references to and from OpenCV functions. For applications that work with high-level features, Darkly provides declassifiers that replace these features with safe representations generated by the appropriate *privacy transforms* (Section VII). Privacy transforms keep the information that applications need for their legitimate functionality while removing the details that may violate privacy.

Inform the user. To help the user balance utility and privacy, our system includes a trusted Darkly console. For applications that operate solely on opaque references, this window is blank. For applications that use declassifiers to access certain input features, it shows to the user the outputs of the privacy transforms being used by the application at any point in time (Section VIII).

The DARKLY console window also contains a *privacy dial* that goes from 0 to 11. By adjusting the dial, the user can increase or decrease the degree of privacy transformation. Even at the setting of 0, DARKLY provides significant privacy protection; in particular, applications are always blocked from directly accessing raw image data.

Be flexible. In rare cases, applications may need to execute arbitrary code on raw inputs. For example, one of our benchmark applications runs the eigenface algorithm [26] to match a face against a database (see Section VI-F).

For such applications, DARKLY provides a special `ibc` language inspired by GNU `bc` [1]. Applications can supply arbitrary `ibc` programs which DARKLY executes internally. These programs are almost pure computations and have no access to the network, system calls, or even system time (Section VI-F). Furthermore, DARKLY restricts their output to 32 bits, thus blocking high-bandwidth covert channels.

VI. IMPLEMENTATION

The prototype implementation of DARKLY consists of approximately 10,000 lines of C/C++ code, not counting the ported `ibc` compiler and VM.

A. OpenCV

OpenCV provides C, C++, and Python interfaces [20] on Windows, Linux, MacOS, iOS and Android. OpenCV is also the default vision library of the Robot Operating System (ROS), a popular platform that runs on 27 robots ranging from the large Willow Garage PR2 to the small iRobot Create or Lego NXT. OpenCV supports diverse input sensors including webcams, Kinects and smart cameras like VC nano 3D[1] or PicSight Smart GigE.[2]

The OpenCV API has more than 500 functions that applications—ranging from interactive art to robotics—use for image-processing and analysis tasks. Our prototype currently supports 145 of these functions (see Section IX for a survey of OpenCV usage in existing applications). Our design exploits both the richness of this API and the fact that individual OpenCV functions encapsulate the minutiae of image processing, relieving applications of the need to access raw image data and helping DARKLY interpose privacy protection in a natural way. That said, the architecture of DARKLY is not specific to OpenCV and can be applied to any perceptual platform with a sufficiently rich API.

OpenCV comprises several components: *libcxcore* implements internal data structures, drawing functions, clustering algorithms, etc.; *libcv* – image processing and computer vision tasks such as image transformations, filters, motion analysis, feature detection, camera calibration, and object detection; *libhighgui* – functions for creating user interfaces; *libml* – machine learning algorithms; *libcvaux* – auxiliary algorithms such as principal component analysis, hidden markov models, view morphing, etc.

OpenCV defines data structures for image data (*IplImage*, *CvMat*, *CvMatND*, etc.), helper data structures (*CvPoint*, *CvRect*, *CvScalar*, etc.), and dynamic data structures (*CvSeq*, *CvSet*, *CvTree*, *CvGraph*, etc.). OpenCV also provides functions for creating, manipulating, and destroying these objects. For example, *cvLoadImage* creates an *IplImage* structure and fills it with the image's pixels and meta-data, while *cvQueryFrame* fetches a frame from a camera or video file and creates an *IplImage* structure with the frame's pixels.

The OpenCV API thus helps developers to program their applications at a higher level. For example, the following 8 lines of C code invert the image and display it to the user until she hits a key:

```
IplImage* img = 0;
// load an image
img=cvLoadImage(argv[1]);
// create a window
cvNamedWindow("mainWin", CV_WINDOW_AUTOSIZE);
cvMoveWindow("mainWin", 100, 100);
// invert the image
cvNot(img, img);
// show the image
cvShowImage("mainWin", img );
// wait for a key
cvWaitKey(0);
// release the image
cvReleaseImage(&img );
```

OpenCV permits only one process at a time to access the camera, thus DARKLY does not allow concurrent execution of multiple applications.

B. Opaque references

To block direct access to raw images, DARKLY replaces pointers to image data with *opaque references* that cannot be dereferenced by applications. Applications can still pass them as arguments into OpenCV functions, which dereference them internally and access the data.

To distinguish opaque references and real pointers, DARKLY exploits the fact that the lower part of the address space is typically reserved for the OS code, and therefore all valid pointers must be greater than a certain value. For example, in standard 32-bit Linux binaries, all valid stack and heap addresses are higher than 0x804800. The values of all opaque references are below this address.

DARKLY cannot simply return an opaque reference in lieu of a pointer to an OpenCV object. Some existing, benign applications do dereference pointers, but only read the metadata stored in the object, not the image data. For example, consider this fragment of an existing application:

```
surfer = cvLoadImage("surfer.jpg",
    CV_LOAD_IMAGE_COLOR);
...
size = cvGetSize(surfer);
```

[1] http://www.vision-components.com/en/products/smart-cameras/vc-nano-3d/

[2] http://www.leutron.com/cameras/smart-gige-cameras/

```
/* create an empty image, same size, depth and
    channels of others */
result = cvCreateImage(size, surfer->depth, surfer
    ->nChannels);
```

Here, *surfer* is an instance of *IplImage* whose meta-data includes the number of channels and the depth of the image. Even though this code does not access the pixel values, it would crash if DARKLY returned an opaque reference instead of the expected pointer to an *IplImage* object.

DARKLY exploits the fact that most OpenCV data structures for images and video include a separate pointer to the actual pixel data. For example, *IplImage*'s data pointer is stored in the *imageData* field; *CvMat*'s data pointer is in the *data* field. For these objects, DARKLY creates a copy of the data structure, fills the meta-data, but puts the opaque reference in place of the data pointer. Existing applications can thus run without any modifications as long as they do not dereference the pointer to the pixels.

C. Interposition

To support unmodified applications, DARKLY must interpose on their calls to the OpenCV library. All of the applications we tested use the dynamically linked version of OpenCV. We implemented DARKLY's interposition layer as a dynamically loaded library and set the LD_PRELOAD shell variable to instruct Linux's dynamic linker to load it before OpenCV. The functions in the interposition library have the same names as the OpenCV functions, thus the linker redirects OpenCV calls made by the application.

This approach works for C functions, but there are several complications when interposing on C++ functions. First, the types of the arguments to DARKLY's wrapper functions must be exactly the same as those of their OpenCV counterparts because the C++ compiler creates new mangled symbols based on both the function name and argument types.

The second, more challenging issue is C++ virtual functions. Because their bindings are resolved at runtime, they are not exported as symbols for the linker to link against. Instead, their addresses are stored in per-object vtables. To interpose on calls to a virtual function, DARKLY overrides the constructor of the class defining the function. The new constructor overwrites the appropriate entries in the vtables of newly created objects with pointers to DARKLY wrappers instead of the genuine OpenCV functions. The formats of objects and vtables are compiler-dependent: for example, GCC stores the vtable address in the object's first 4 bytes. Our code for hooking vtables is as follows:[3]

```
extern "C" void patch_vtable(void *obj, int
    vt_index, void *our_func) {
  int* vptr = *(int**)obj;
  // align to page size:
  void* page = (void*)(int(vptr) & ~(getpagesize()
      -1));
  // make the page with the vtable writable
  mprotect(page, getpagesize(), PROT_WRITE|
      PROT_READ))
  vptr[vt_index] = (int)our_func;
}
```

The *vt_index* parameter specifies the index of the vtable entry to be hooked. GCC creates vtable entries in the order of the virtual function declarations in the class source file.

Dispatching OpenCV functions. For each call made by an application to an OpenCV function, the interposition library must decide whether to execute it within the application or forward it to the trusted DARKLY server running as a separate "user" on the same device (only this server has access to camera inputs). To complicate matters, certain OpenCV functions accept variable-type arguments, e.g., *cvNot* accepts either *IplImage*, or *CvMat*. OpenCV detects the actual type at runtime by looking at the object's header.

After intercepting a call to an OpenCV function, the interposition library determines the type of each argument and checks whether it contains an opaque reference (the actual check depends on the object's type). If there is at least one argument with an opaque reference, executing the function requires access to the image. The interposition library marshals the local arguments and opaque references, and forwards the call to DARKLY for execution.

If none of the arguments contain an opaque reference, the function does not access the image and the interposition library simply calls the function in the local OpenCV library.

D. Privacy transforms

For applications that need access to image features—for example, to detect motion, track certain objects, etc.—DARKLY provides declassifier functions. Our prototype includes the following declassifiers: *cvMoments* returns moments, *cvFindContours* – contours, *cvGoodFeaturesToTrack* – sets of corner points, *cvCalcHist* – pixel histograms, *cvHaarDetectObjects* – bounding rectangles for objects detected using a particular model (DARKLY restricts applications to predefined models shipped with OpenCV), *cvMatchTemplate* – a map of comparison results between the input image and a template, *cvGetImageContent* – image contents (transformed to protect privacy).

Declassifier	Privacy transform
cvMoments	Sketching
cvFindContours	Sketching
cvGoodFeaturesToTrack	Increasing feature threshold
cvCalcHist	Sketching
cvHaarDetectObjects	Generalization
cvMatchTemplate	Thresholding match values
cvGetImageContent	Thresholding binary image

Table I
TRANSFORMS USED FOR EACH DARKLY DECLASSIFIER.

[3] Cf. http://www.yosefk.com/blog/machine-code-monkey-patching.html

Declassifiers apply an appropriate privacy transform (see Section VII) to the input, as shown in Table I. For example, *cvGetImageContent* returns a thresholded binary representation of the actual image. Furthermore, these outputs are displayed on the Darkly console to inform the user.

E. Trusted services

Trusted services in Darkly enable the application to send data to the user without actually "seeing" it.

Trusted display. The trusted display serves a dual purpose: (1) an application can use it to show images to which it does not have direct access, and (2) it shows to the user the privacy-transformed features and objects released to the application by declassifiers (see Section VIII).

We assume that the OS blocks the application from reading the contents of the trusted display via "print screen" and similar system calls. These contents may also be observed and recaptured by the device's own camera. We treat this like any other sensitive item in the camera's field of vision (e.g., contents of an unrelated computer monitor).

To enable applications to display images without access to their contents, Darkly must interpose on HighGUI, OpenCV's user interface (UI) component [13]. HighGUI is not as extensive as some other UI libraries such as Qt, but the general principles of our design are applicable to any UI library as long as it is part of the trusted code base. Among other things, HighGUI supports the creation and destruction of windows via its *CvNamedWindow* and *CvDestroyWindow* functions. Applications can also use *cvWaitKey* to receive keys pressed by the user, *cvSetMouseCallback* to set custom callback functions for mouse events, and *cvCreateTrackbar* to create sliders and set custom handlers.

The interposition library forwards calls to any of these functions to Darkly. For functions like *CvNamedWindow*, Darkly simply calls the corresponding OpenCV function, but for the callback-setting functions such as *cvSetMouseCallback* and *cvCreateTrackbar*, Darkly replaces the application-defined callback with its own function. When the Darkly callback is activated by a mouse or tracker event, it forwards these events to the interposition library, which in turns invokes the application-defined callback.

User input may be privacy-sensitive. For example, our benchmark OCR application recognizes characters drawn by the user using the mouse cursor. Darkly replaces the actual mouse coordinates with opaque references before they are passed to the application-defined callback.

HighGUI event handling is usually synchronous: the application calls *cvWaitKey*, which processes pending mouse and tracker events and checks if any key has been pressed. This presents a technical challenge because most application-defined callbacks invoke multiple OpenCV drawing functions. If callback interposition is implemented synchronously, i.e., if the Darkly callback handler forwards the event to the application-defined callback and waits for it to finish, the overhead of interposition (about 9% per each call forwarded over an interprocess socket, in our experiments) increases linearly with the number of OpenCV functions invoked from the application-defined callback. In practice, this causes the OpenCV event buffer to overflow and start dropping events.

Instead, our callback handler runs in a separate thread in the Darkly server. The interposed callbacks forward GUI events asynchronously to a thread in the interposition library, which then invokes the application-defined callbacks. Because most OpenCV functions are not thread-safe, we serialize access with a lock in the interposition library.

```
void on_mouse( int event, int x, int y, int flags,
     void* param ) {
  ...
  cvCircle(imagen, cvPoint(x,y), r, CV_RGB(red,
      green,blue), -1, 4, 0);
  // Get clean copy of image
  screenBuffer=cvCloneImage(imagen);
  cvShowImage( "Demo", screenBuffer );
  ...
}

int main(int argc, char** argv ) {
  ...
  cvSetMouseCallback("Demo",&on_mouse, 0 );
  for (;;) { ... c = cvWaitKey(10); ... } }
}
```

Listing 2. Sample callback code.

Trusted storage. To store images and video without accessing their contents, applications can invoke *cvSaveImage* or *cvCreateVideoWriter*. The interposition library forwards these calls to Darkly, which redirects them to system-configured files that are owned and accessible only by the user who is running Darkly. Dropbox or Google Drive can be mounted as (user-controlled) remote file systems.

With this design, an application cannot store data into its own files, while standard OS file permissions block it from reading the user's files.

F. Support for application-provided code

Even though the OpenCV API is very rich, some applications may need to run their own computations on raw images rather than chain together existing OpenCV functions. Darkly provides a special-purpose language that application developers can use for custom image-processing programs. Darkly executes these programs inside the library on the true image data (as opposed to privacy-preserving representations returned by the declassifiers), but treats them as untrusted, potentially malicious code. Isolating arbitrary untrusted programs is difficult, but our design takes advantage of the fact that, in our case, these *domain-specific* programs deal solely with image processing.

The Darkly language for application-supplied untrusted computations is called `ibc`. It is based on the GNU `bc` language [1]. We chose `bc` for our prototype because it (1) supports arbitrary numerical computations but has no OS interface, (2) there is an existing open-source implementation, and (3) its C-like syntax is familiar to developers. ibc programs cannot access DARKLY's or OpenCV's internal state, and can only read or write through the DARKLY functions described below. They do not have access to the network or system timers, minimizing the risk of covert channels, and are allowed to return a single 32-bit value.[4]

ibc compiler. The GNU bc compiler takes a source file as input, generates bytecode, and executes it in a bytecode VM. DARKLY cannot pay the cost of bytecode generation every time an application executes the same program (for example, for each frame in a video). Therefore, we separated the bytecode generator and the VM.

DARKLY adds a *bcCompile* function to the OpenCV API. It takes as input a string with ibc source code and returns a string with compiled bytecode. DARKLY also adds a *cvExecuteUntrustedCode* function, which takes a bytecode string and pointers to OpenCV objects, executes the bytecode on these objects, and returns a 32-bit value to the application. The latter required a VM modification because GNU bc does not allow the main program to return a value.

To support computations on images and matrices, DARKLY adds *iimport* and *iexport* functions. *iimport* takes the id of an OpenCV object (i.e., the order in which it was passed to *cvExecuteUntrustedCode*), x and y coordinates, and the byte number, and returns the value of the requested byte of the pixel at the x/y position in the image. Similarly, *iexport* lets an ibc program to set pixel values.

Using custom ibc programs. To illustrate how to write custom image-processing code in ibc, we modified an existing application that inverts an image by subtracting each pixel value from 255 (this can be done by calling OpenCV's *cvNot* function, but this application does not use it):

```
img = cvLoadImage(argv[1], 1);
data = (uchar *)img->imageData;
// invert the image
for(i=0;i<img->height;i++)
    for(j=0;j<img->width;j++)
      for(k=0;k<channels;k++)
        data[i*step+j*channels+k]=255-data[i*step+
            j*channels+k];
```

Listing 3. Application code for inverting an image.

```
bc_invrt_tmpl =
"for (i=0; i<%d;i++) {
    for (j=0; j<%d; j++) {
      for (k=0; k<4; k++) {
            v = iimport(0, i, j, k);
            iexport(0, i, j, k, 255-v); } } }
 return 0;";
img = cvLoadImage(argv[1], 1);
snprintf(bc_invert_code, MAX_SIZE, bc_invert_tmpl,
     img->height, img->width);
bc_bytecode = bcCompile(bc_invert_code);
ret = cvExecuteUntrustedCode(bc_bytecode, img, 0,
     0);
```

Listing 4. Using ibc code for inverting an image.

The *iimport*/*iexport* interface can also be used to access any 1-, 2- or 3-D array. For example, we took an existing face recognition application (see Section IX) and wrote an ibc program to find the closest match between the input face's eigen-decomposition coefficients computed by *cvEigenDecomposite* and a dataset of faces. Running this program inside DARKLY allows the application to determine whether a match exists without access to the actual eigen-decomposition of the input face. The code is shown below.

```
int findNearestNeighbor( const Eigenface& data,
    float * projectedTestFace ) {
    double leastDistSq = 999999999; //DBL_MAX;
    int iNearest = 0;

    for( int iTrain = 0; iTrain < data.nTrainFaces
        ; iTrain++ ) {
        double distSq = 0;
        for( int i = 0; i < data.nEigens; ++i ) {
            float d_i = projectedTestFace[i] -
                data.projectedTrainFaceMat->data.
                fl[iTrain * data.nEigens + i];
            distSq += d_i * d_i / data.eigenValMat
                ->data.fl[i]; }
        if( distSq < leastDistSq ) {
            leastDistSq = distSq;
            iNearest = iTrain; } }
    return iNearest;
}

cvEigenDecomposite(image,
           data.nEigens,
           &(*( data.eigenVectVec.begin())),
           0, 0, data.pAvgTrainImg,
           projectedTestFace);
int iNearest = findNearestNeighbor(data,
    projectedTestFace);
```

Listing 5. Part of face-recognition application code for calculating the closest match to the input image.

```
bc_dist_tmpl =
"fscale=2;
leastdistsq = 999999999
inearest = -1
for( itrain = 0; itrain < %d; itrain++ ) {
    distsq = 0.0;

    for( i = 0; i < %d; ++i ) {
           a = iimport(0, i, 0, 0)
           b = iimport(1, itrain * 2 + i, 0, 0)
           di = a-b
           c = iimport(2,i,0,0);
           distsq += di * di / c ;
    }
    if( distsq < leastdistsq ) {
          leastdistsq = distsq;
          inearest = itrain;
    }
}
return inearest;";

cvEigenDecomposite(image,
           data.nEigens,
```

[4] The current DARKLY prototype allows an application to gain more information by invoking ibc programs multiple times, but it is easy to restrict the number of invocations if needed.

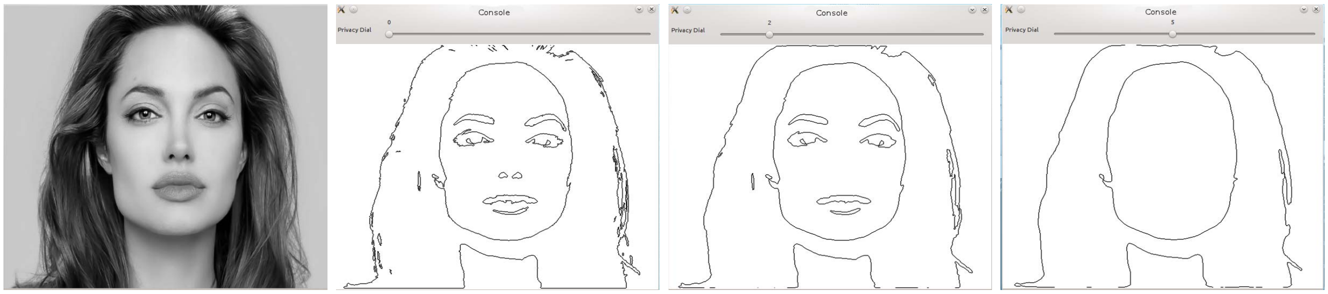


Figure 2. Output of the sketching transform on a female face image at different privacy levels.

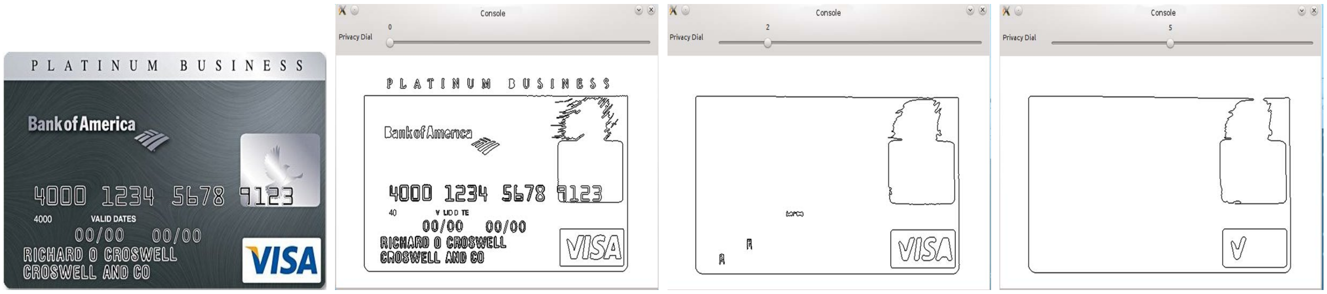


Figure 3. Output of the sketching transform on a credit card image at different privacy levels.

```
            &(*( data.eigenVectVec.begin())),
            0, 0, data.pAvgTrainImg,
            projectedTestFace);

snprintf(bc_dist_code, MAX_SIZE, bc_invert_tmpl,
    data.nTrainFaces, data.nEigens);
bc_bytecode = bcCompile(bc_dist_code);
int iNearest = cvExecuteUntrustedCode(bc_bytecode,
     projectedTestFace, data.projectedTrainFaceMat
    , data.eigenValMat);
```

Listing 6. Modified face-recognition application code using `ibc` for calculating the closest match to the input image.

VII. Privacy transforms

In Section IX, we show that many OpenCV applications can work, without any modifications, on opaque references. Some applications, however, call OpenCV functions like *cvMoments*, *cvFindContours*, or *cvGoodFeaturesToTrack* which return information about certain features of the image. We call these functions *declassifiers* (Section VI-D).

To protect privacy, declassifiers transform the features before releasing them to the application. The results of the transformation are shown to the user in the Darkly console window (Section VIII). The user can control the level of transformation by adjusting the privacy dial on this screen.

The transformations are specific to the declassifier but application-independent. For example, the declassifier for *cvGetImageContent* replaces the actual image with a thresholded binary representation (see Fig. 7). The declassifier for *cvGoodFeaturesToTrack*, which returns a set of corner points, applies a higher *qualitylevel* threshold as the dial setting increases, thus only the strongest candidates for corner points are released to the application.

The declassifiers for *cvFindContours*, *cvMoments*, and *cvCalcHist* apply the sketching transform from Section VII-A to the image before performing their main operation (e.g., finding contours) on the transformed image. The application thus obtains only the features such as contours or moments and not any other information about the image.

Applying a privacy transform does not affect the accuracy of OpenCV functions other than the declassifiers because these functions operate on true, unmodified data.

A. Sketching

The *sketch* of an image is intended to convey its high-level features while hiding more specific, privacy-sensitive details. A loose analogy is publicly releasing statistical aggregates of a dataset while withholding individual records.

The key to creating sketches is to find the contours of the image, i.e., the points whose greyscale color value is equal to a fixed number. In our prototype we use a hardcoded value of 50% (e.g., 127 for 8-bit color). Contours by themselves don't always ensure the privacy properties we want. For example, in Fig. 3, contours reveal a credit card number. Therefore, the sketching transform uses contours in combination with two types of low-pass filters.

First, the image is blurred[5] *before* contour detection. Blurring removes small-scale details while preserving large-scale features. The privacy dial controls the size of the filter

[5] We use a box filter because it is fast: it averages the pixels in a box surrounding the target pixel. We could also use a Gaussian or another filter.

kernel. Higher kernel values correspond to more blurring and fewer details remaining after contour detection.

Just as contour detection alone is insufficient, low-pass filtering alone would have been insufficient. For example, image deblurring algorithms can undo the effect of box filter and other types of blur; in theory, this can be achieved exactly as long as the resolution of the output image is not decreased [15]. By returning only the contours of the blurred image, our sketching transform ensures that blurring cannot be undone (it also removes all contextual information).

Another low-pass filter is applied *after* contour detection. The transform computes the mean radius of curvature of each contour (suitably defined for nondifferentiable curves on discrete spaces) and filters out the contours whose mean radius of curvature is greater than a threshold. The threshold value is controlled by the privacy dial. Intuitively, this removes the contours that are either too small or have too much entropy due to having many "wrinkles."

Reducing an image to its contours, combined with low-pass filtering, ensures that not much information remains in the output of the transform. Due to blurring, no two contour lines are too close to each other, which upper-bounds the total perimeter of the contours in an image of a given size.

Fig. 4 illustrates how sketching reduces information available to the application, as a function of the user-selected privacy level. We also experimentally estimated the entropy of sketches on a dataset of 30 frontal face images sampled from the Color FERET database.[6] These were cropped to the face regions, resulting in roughly 220x220 images. We can derive an upper bound on entropy by representing contours as sequences of differences between consecutive points, which is a more compact representation. Fig. 5 shows that, for reasonable values of the privacy dial (3–6), the resulting sketches can be represented in 500-800 bytes.

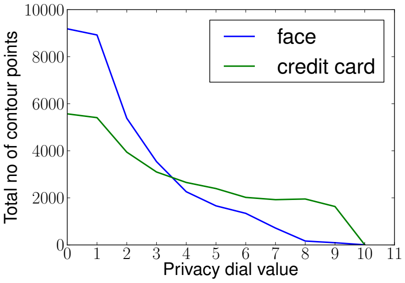


Figure 4. Sketching: reduction in information available to the application for images from Figs. 2 and 3.

B. Generalization

In addition to generic image manipulation and feature extraction functions like *cvFindContours*, OpenCV also provides model-based object detectors. An application can load a Haar classifier using *cvLoadHaarClassifierCascade* and detect objects of a certain class (for example, faces) by calling *cvHaarDetectObjects* with a class-specific model. To prevent applications from inferring information via malicious models, the current DARKLY prototype only allows predefined models that ship with OpenCV.

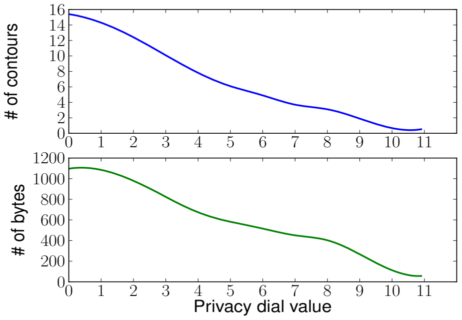


Figure 5. Sketching: reduction in average information available to the application for facial images in FERET database (size roughly 220x220).

If a match is found, *cvHaarDetectObjects* returns a rectangular bounding box containing the object, but not the pixels inside the box. This still carries privacy risks. For example, an application that only has an opaque reference to the box containing a face can use OpenCV calls to detect the location of the nose, mouth, etc. and learn enough information to identify the face. To prevent this, DARKLY applies a generalization-based privacy transform.

Face generalization. Generalization has a long history in privacy protection; we explain our approach using face detection as an example. Our privacy transform replaces the actual face returned by *cvHaarDetectObjects* with a "generic" face selected from a predefined, model-specific dictionary of canonical face images. We call our face generalization algorithm *cluster–morph*.

The generalization idiom is already familiar to users from "avatars" in video games, online forums, etc. Sometimes avatars are picked arbitrarily, but often users choose an avatar that best represents their own physical characteristics. In the same way, the generalized face in DARKLY is intended to be perceptually similar to the actual face, although, unlike an avatar, it is programmatically generated.

There are two components to generalization: first, fixing (and if necessary, pre-processing) the canonical dictionary, and second, choosing a representative from this dictionary for a given input face. The former is a one-time process, the latter is part of the transform. For the first component, one straightforward approach is to simply pick a small dictionary of (say) 20 faces and run a face detector on the actual face to find and return its closest match from the dictionary.

Our proposed *cluster–morph* technique is a promising but more complex approach to generalization. It works as follows: start from a large database of images and compute its eigenfaces by applying a well-known algorithm [26] that uses Principal Component Analysis to calculate a set of

[6] http://www.nist.gov/itl/iad/ig/colorferet.cfm

basis vectors for the set of all faces. Then compute the eigen-decomposition of each face, i.e., represent it as a linear combination of the basis vectors, and truncate each decomposition to the first (say) 30 principal components. Next, cluster the set of faces using the Euclidean distance between decompositions as the distance function.

Finally, to find the canonical face "representing" each cluster, *morph* the faces in the cluster using standard morphing algorithms [3]. Fig. 6 shows an example from a cluster of size 2 obtained by hierarchical clustering on a 40-person ORL dataset [22]. Clustering and morphing are done once to produce a fixed dictionary of canonical faces.

We propose to use hierarchical agglomerative clustering. It offers the key advantage that the level of generalization can be adjusted based on the setting of the privacy dial: as the dial value increases, the transform selects clusters higher in the hierarchy. If all clusters have at least k elements, then the number of clusters is no more than $\frac{2N}{k}$ where N is the total number of faces in the database.

At runtime, to generalize a given input face, compute its eigen-decomposition, calculate its distance to each cluster center,[7] and pick the closest. The transform then returns the morphed image representing this cluster to the application.

Our DARKLY prototype includes a basic implementation of *cluster–morph*. Evaluating the algorithm on the Color FERET database is work in progress. There are at least three challenges: measuring the effectiveness of face clustering, finding a mapping between privacy dial values and cluster hierarchy levels (e.g., dial values can be pegged to either cluster sizes or cluster cohesion thresholds), and developing metrics for quantifying privacy protection.

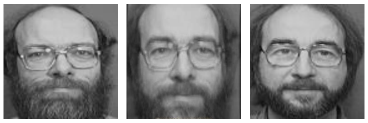


Figure 6. Face morphing for generalization. The left and right faces belong to the same cluster; the morph "representing" this cluster is in the center.

Our *cluster–morph* algorithm is inspired in part by Newton et al.'s algorithm for k-anonymity-based facial de-identification [19], which works as follows: given a database of images, repeatedly pick a yet-unclustered image from the database and put it in a cluster with $k-1$ of its "closest" images, according to an eigenface-based distance measure. For each face in the input database, the average of the faces in its cluster constitutes its de-identified version.

The salient differences in our case are as follows: our goal is not k-anonymity within a database, but finding a canonical representation w.r.t. a globally predefined dataset (in particular, the input image is *not* drawn from this dataset). Further, Newton et al.'s algorithm has some weaknesses for our purposes: it uses greedy clustering instead of more principled methods, requires re-clustering if the privacy dial changes, and, finally, in our experiments averaging of faces produced results that were visually inferior to morphing.

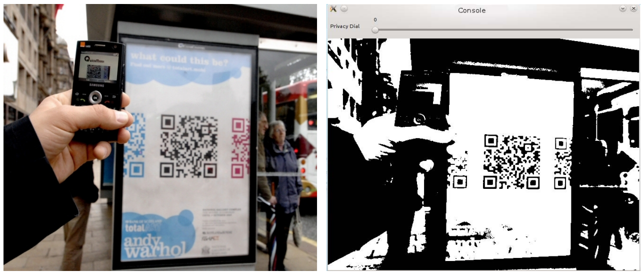


Figure 7. Output of the thresholding binary transform on an image of a street scene with a QR code. QR decoding application works correctly with the transformed image.

VIII. DARKLY CONSOLE

The DARKLY console is a DARKLY-controlled window that shows a visual representation of the features and objects returned to the application by the declassifiers. For applications that operate exclusively on opaque references, the DARKLY console is blank. For applications that use declassifiers, the DARKLY console shows the outputs of the corresponding privacy transforms—see examples in Figs. 8 and 9. We assume that this window cannot be spoofed by the application. In general, constructing trusted UI is a well-known problem in OS design and not specific to DARKLY.

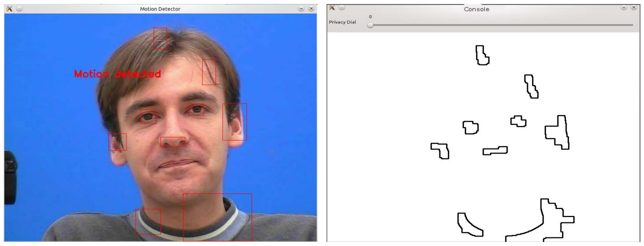
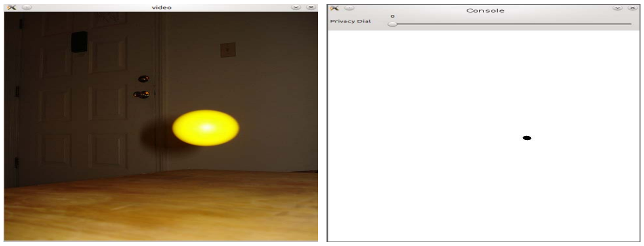


Figure 8. Motion detector: actual image and the DARKLY console view. Application works correctly with the transformed image.

The DARKLY console is implemented as a separate process communicating with DARKLY over UNIX domain sockets. With this design, the application's declassifier function calls need not be blocked until the DARKLY console has finished rendering. We did not implement the DARKLY console as a thread inside the DARKLY server because both use OpenCV, and OpenCV functions are not thread-safe.

Consecutive DARKLY console views are stored as a movie file in AVI or MPG format. If storage is limited, they can be compressed and/or stored at reduced resolution. The user can

[7] A cluster center is the mean of the eigen-decomposites of each image in the cluster. It does not correspond to the morphed image. Since eigen-decomposition of a face is a linear transformation, averaging in the eigenspace is the same as averaging in the original space; thus, the image corresponding to the cluster center is a plain pixelwise average of the faces in the cluster. This average would be unsuitable as a canonical representative due to artifacts such as *ghosting*, which is why we use the morphed image.



Privacy Dial 0

Figure 9. Ball tracker: actual image and the DARKLY console view. Application works correctly with the transformed image.

play back the movie and see how the information released to the application by privacy transforms evolved over time.

Privacy dial. The DARKLY console includes a slider for adjusting the level of transformation applied by the privacy transforms. The values on the slider range from 0 to 11. Absolute values are interpreted differently by different transforms, but higher values correspond to coarser outputs (more abstract representations, simpler contours, etc.). For example, higher values cause the sketching declassifier to apply a larger box filter to smoothen the image before finding the contours, thus removing more information (see Fig. 3).

IX. EVALUATION

We evaluated DARKLY on 20 OpenCV applications, listed in Table II along with their source URLs. These applications have been selected from Google Code, GitHub, blogs, and OpenCV samples for the variety and diversity of their features and the OpenCV functionality they exercise. With the exception of OCR, which uses the C++ interface for nearest-neighbor clustering, they use OpenCV's C interface.

Our DARKLY prototype is based on OpenCV release 2.1.0. Applications were evaluated on a Segway RMP-50 robot running ROS Fuerte and/or a laptop with a quad-core 2.40GHz Intel Core i3 CPU and 4 GB of RAM running 32 bit Ubuntu 11.10 desktop edition.

Results are summarized in Table III. 18 out of 20 applications required *no modifications* to run on DARKLY, except very minor formatting tweaks in a couple of cases (removing some header files so that the program compiles in Linux). For the face recognizer, we re-implemented the eigenface matching algorithm in our `ibc` language (see Section VI-F) so that it can run on true images inside the library, returning only the match/no match answer to the application.

For all tests, we used either a benchmark video dataset of a person talking,[8] or the sample images and videos that came with the applications, including OpenCV sample programs.[9] Depending on the application, frame rates were computed for the video or over the input images.

Application	URL
OCR for hand-drawn digits	http://blog.damiles.com/2008/11/basic-ocr-in-opencv/
Security cam	http://code.google.com/p/camsecure/
Ball tracker	https://github.com/liquidmetal/AI-Shack--Tracking-with-OpenCV/blob/master/TrackColour.cpp
QR decoder	https://github.com/josephholsten/libdecodeqr
PrivVideo, video background subtractor and streamer	http://theembeddedsystems.blogspot.com/2011/05/background-subtraction-using-opencv.html
Facial features detector	http://opencvfacedetect.blogspot.com/2010/10/face-detectionfollowed-by-eyesnose.html
Face recognizer	http://www.cognotics.com/opencv/servo_2007_series/index.html
Histogram calculator (RGB)	http://www.aishack.in/2010/07/drawing-histograms-in-opencv/
Histogram calculator (Hue-Saturation)	http://opencv.willowgarage.com/documentation/cpp/histograms.html
Square detector	https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/squares.c?rev=27
Morphological transformer	https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/morphology.c?rev=27
Intensity/contrast changer for images/histograms	https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/demhist.c?rev=1429
Pyramidal downsampler + Canny edge detector	http://dasl.mem.drexel.edu/∼noahKuntz/openCVTut1.html
Image adder	http://silveiraneto.net/2009/12/08/opencv-adding-two-images/
H-S histogram back-projector	http://dasl.mem.drexel.edu/∼noahKuntz/openCVTut6.html
Template matcher	http://opencv.willowgarage.com/wiki/FastMatchTemplate?action=AttachFile&do=view&target=FastMatchTemplate.tar.gz
Corner finder	http://www.aishack.in/2010/05/corner-detection-in-opencv/
Hand detector	http://code.google.com/p/wpi-rbe595-2011-machineshop/source/browse/trunk/handdetection.cpp
Laplace edge detector	https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/laplace.c?rev=27
Ellipse fitter	https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/fitellipse.c?rev=1429

Table II
BENCHMARK OPENCV APPLICATIONS.

Performance. Performance is critically important for perceptual applications that deal with visual data. If the overhead of privacy protection caused frame rates to drop too much, applications would become unusable. Figure 10 shows that the performance overhead of DARKLY is very minor and, in most cases, not perceptible by a human user.

The effect of a given privacy transform depends on the setting of the privacy dial, aka the privacy level. For example, sketching, the transform for the *cvFindContours* declassifier, applies different amounts of blurring before finding contours. Fig. 11 shows that the performance variation of the security camera application at different privacy levels is minimal (within 3%). Interestingly, in this case performance does not change monotonically with the privacy level. The reason is that the OpenCV function used by the sketching transform switches algorithms depending on the parameters.

Tradeoffs between privacy and utility. Table III shows that for most applications, there is *no change of functionality* and *no loss of accuracy* even at the maximum privacy setting.

[8] http://www-prima.inrialpes.fr/FGnet/data/01-TalkingFace/talking_face.html

[9] https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c?rev=27

Application	**LoC**	**Modified LoC**	**Change in functionality**	**Information accessed**
QR decoder	4700	19	Works only at privacy level 0 *	Contours, thresholded image
Face recognizer	851	1 + 19 (ibc)	No change	Match/no match
OCR	513	0	No change	Output digit
Template matcher	483	0	No change	Match matrix
Security cam	312	0	See Fig. 12	Contours
Facial features detector	258	0	No change **	Rectangular bounding boxes
Square detector	238	0	See Fig. 12	Contours
Ellipse fitter	134	0	See Fig. 12	Contours
Intensity/contrast changer for images/histograms	127	0	No change	Histograms
Ball tracker	114	0	See Fig. 12	Moments
PrivVideo	96	0	No change	None
Morphological transformer	91	0	No change	None
H-S histogram backprojector	81	0	See Fig. 12	Histogram
Laplace edge detector	73	0	No change	None
RGB histogram calculator	70	0	See Fig. 12	Histogram
H-S histogram calculator	58	0	See Fig. 12	Histogram
Hand detector	48	0	No change	Yes/no
Corner finder	42	0	See Fig. 12	Corner coordinates
Image adder	37	0	No change	None
Downsampler + Canny edge detector	36	0	No change	None

* Even at level 0, privacy from the QR decoder is protected by the thresholding binary transform.
** Feature detection is performed on privacy-transformed faces (Section VII-B).

Table III
EVALUATION OF DARKLY ON OPENCV APPLICATIONS.

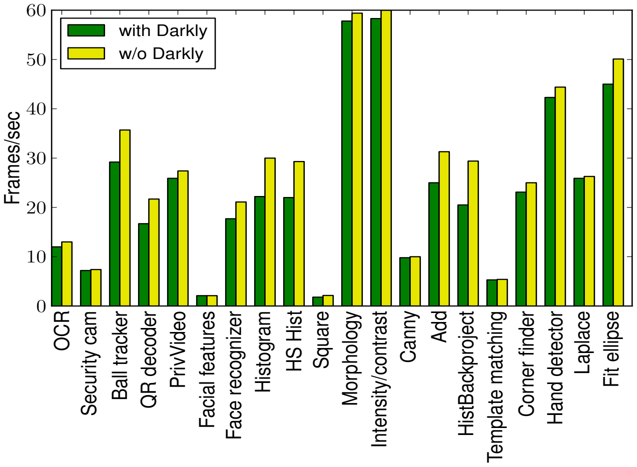


Figure 10. Frame rates with and without DARKLY.

The reason is that these applications do not access raw images and can operate solely on opaque references.

One application, the QR decoder, works correctly at privacy level 0, but not at higher settings. Even at privacy level 0, significant protection is provided by the thresholding binary transform (see Fig. 7). For the remaining applications, the tradeoff between their accuracy and user-selected privacy level is shown in Fig. 12.

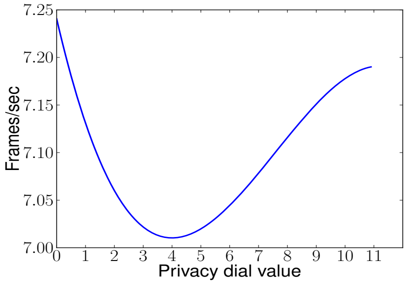


Figure 11. Frame rate of the security-camera application as a function of the privacy level. At levels above 4, OpenCV switches from directly calculating the convolution to a DFT-based algorithm optimized for larger kernels. Furthermore, as privacy level increases, smaller motions are not detected and the application has to process fewer motions.

Support for other OpenCV applications. We found 281 GitHub projects mentioning "vision," "applications," and "opencv."[10] Filtering out empty projects and clones with the same name and codebase reduced the set to 77 projects.

[10] A simple search for "opencv" returns different parts of the OpenCV library itself and does not work for finding OpenCV applications.

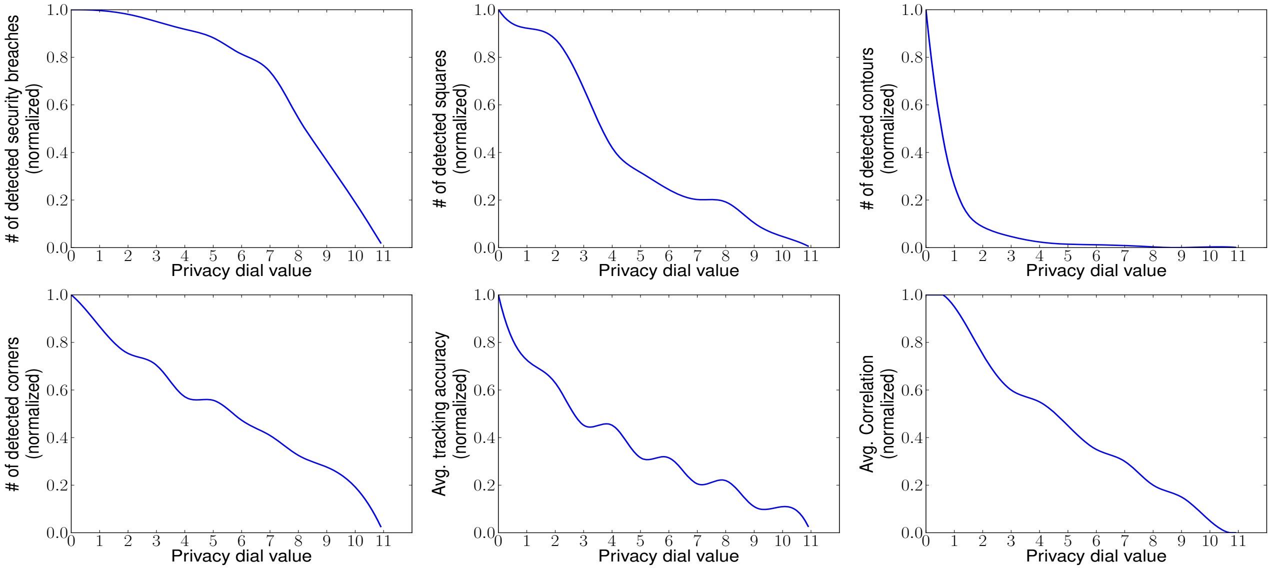


Figure 12. Change in the number of detected security breaches (Security cam), detected squares (Square detector), detected contours (Ellipse fitter), moments (Ball tracker), and histograms (RGB and H-S histogram calculators, Intensity/contrast changer for images/histograms, and H-S histogram backprojector) as the privacy level increases. Correlation between histograms was calculated using the cvHistCompare function. Accuracy for tracking was measured using the Euclidean distance between the object's original position and the reported position after applying privacy transforms.

We scanned these 77 projects for invocations of *cvGet2D*, *cvGetAt*, or *cvGetRawData*, and direct accesses to the *imageData* field of the image data structure. After removing the spurious matches caused by included OpenCV header files, we found that 70% of the projects (54 out of 77) do not access raw pixels. Furthermore, only 11 projects access the network, and only 2 access audio inputs.

These 77 projects call a total of 291 OpenCV functions, of which 145 are already supported by our DARKLY prototype, 118 can be supported with opaque references, 15 can be supported with the sketching-based declassifier, and 3 require porting application code to `ibc`. These 281 functions are sufficient to support 68 of the 77 surveyed projects.

The remaining 9 projects make calls to unsupported OpenCV functions (10 in total) that perform tasks such as optical flow (*cvCalcOpticalFlowBM*, *cvCalcOpticalFlowHS* *cvCalcOpticalFlowLK*, and *cvCalcOpticalFlowPyrLK*), object tracking (*cvCamShift*, *cvMeanShift*, and *cvSnakeImage*), camera calibration (*ComputeCorrespondEpilines*), motion analysis (*cvSegmentMotion*), and image segmentation (*cvWatershed*). Supporting these functions in DARKLY would require new, task-specific privacy transforms and is an interesting topic for future research.

X. RELATED WORK

Denning et al. [7] showed that many off-the-shelf consumer robots do not use proper encryption and authentication, thus a network attacker can control the robot or extract sensitive data. By contrast, DARKLY protects users from untrusted applications running on a trusted robot. PlaceRaider [25] is a hypothetical mobile malware that can construct a 3-D model of its environment from phone-camera images. DARKLY prevents this and similar attacks.

SciFi [21] uses secure multiparty computation to match faces against a database. Matching takes around 10 seconds per image, thus SciFi is unusable for real-time applications. The threat model of DARKLY is different (protecting images from untrusted applications), it handles many more perceptual tasks, and can protect real-time video feeds.

Ad-hoc methods for protecting specific sensitive items include the blurring of faces and license plates in Google Maps' Street View [24]. Senior et al. [23] suggested image segmentation to detect sensitive objects in surveillance videos and transform them according to user-provided policies. To protect surveillance videos on the network, Dufaux and Ebrahimi [8] proposed to encrypt regions of interest. This requires computationally expensive, offline image segmentation and it is not clear whether perceptual applications would work with the modified videos. Chan et al. [5] developed a method for counting the number of pedestrians in surveillance videos without tracking any single individual.

Sweeney et al. published several papers [11, 12, 19] on "de-identifying" datasets of face images. Many of their techniques, especially in the k-same-Eigen algorithm, are similar to the generalization transform described in Section VII-B. They do a "greedy" version of clustering and their model-based face averaging has similarities with face morphing.

Showing the outputs of privacy transforms to the user on the DARKLY console is conceptually similar to the *sensor-access widgets* by Howell and Schechter [14]. Their widgets, however, display the entire camera feed because applications in their system have unrestricted access to visual inputs.

Augmented Reality (AR) applications are a special subset of perceptual applications that not only read perceptual data but also modify and display some parts of the input back to the user. To protect user privacy from such applications, D'Antoni et al. [6] argue that the OS should provide new higher-level abstractions for accessing perceptual data instead of the current low-level sensor API. Jana et al. [16] built a new OS abstraction (*recognizers*) and a permission system for enforcing fine-grained, least-privilege access to perceptual data by AR applications. This permission-based approach is complementary to DARKLY.

XI. Future work

DARKLY is the first step towards privacy protection for perceptual applicatons. Topics for future research include: (1) evaluation of functionality and usability on a variety of computer-vision tasks, (2) support for application-provided, potentially untrusted object recognition models (the current transform for *cvHaarDetectObjects* is based on the face detection model shipped with OpenCV) and third-party object recognition services such as Dextro Robotics, and (3) development of privacy transforms for untrusted, application-provided image-processing code. The latter may obviate the restriction on the outputs of untrusted code, but would also require a new visualization technique for displaying these outputs to the user on the DARKLY console.

Longer-term research includes: (4) preventing inferential leaks by using large-scale, supervised machine learning to construct detectors and filters for privacy-sensitive objects and scenes, such as certain text strings, gestures, patterns of movement and physical proximity, etc., and (5) extending the system to other perceptual inputs such as audio.

Acknowledgments. We are grateful to David Molnar, Scott Saponas, and Ryan Calo for helpful discussions during the early part of this work and to Piyush Khandelwal for helping us evaluate DARKLY on the Segway RMP-50 robot.

This work was supported by the NSF grant CNS-0746888, the MURI program under AFOSR Grant No. FA9550-08-1-0352, and Google PhD Fellowship to Suman Jana.

References

[1] bc Command Manual. http://www.gnu.org/software/bc/manual/html_chapter/bc_toc.html.

[2] R. V. Bruegge. Facial Recognition and Identification Initiatives. http://biometrics.org/bc2010/presentations/DOJ/vorder_bruegge-Facial-Recognition-and-Identification-Initiatives.pdf, 2010.

[3] T. Bui, M. Poel, D. Heylen, and A. Nijholt. Automatic face morphing for transferring facial animation. In *CGIM*, 2003.

[4] M. R. Calo. People can be so fake: A new dimension to privacy and technology scholarship. *Penn St. L. Rev.*, 114:809, 2009.

[5] A. Chan, Z. Liang, and N. Vasconcelos. Privacy preserving crowd monitoring: Counting people without people models or tracking. In *CVPR*, 2008.

[6] L. D'Antoni, A. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, M. Veanes, and H. Wang. Operating system support for augmented reality applications. Technical Report MSR-TR-2013-12, Microsoft Research.

[7] T. Denning, C. Matuszek, K. Koscher, J. Smith, and T. Kohno. A Spotlight on Security and Privacy Risks with Future Household Robots: Attacks and Lessons. In *Ubicomp*, 2009.

[8] F. Dufaux and T. Ebrahimi. Scrambling for video surveillance with privacy. In *CVPRW*, 2006.

[9] C. Dwork. Differential privacy. In *ICALP*, 2006.

[10] C. Dwork. A firm foundation for private data analysis. In *CACM*, 2011.

[11] R. Gross, E. Airoldi, B. Malin, and L. Sweeney. Integrating utility into face de-identification. In *PET*, 2006.

[12] R. Gross, L. Sweeney, F. De la Torre, and S. Baker. Model-based face de-identification. In *CVPRW*, 2006.

[13] HighGUI: High-level GUI and Media I/O. http://opencv.willowgarage.com/documentation/python/highgui__high-level_gui_and_media_i_o.html.

[14] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, camera, and other sensors. In *W2SP*, 2010.

[15] R. Hummel, B. Kimia, and S. Zucker. Deblurring gaussian blur. *CVGI*, 1987.

[16] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. Technical Report MSR-TR-2013-11, Microsoft Research.

[17] D. McCullagh. Call it Super Bowl Face Scan I. http://www.wired.com/politics/law/news/2001/02/41571, 2001.

[18] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[19] E. Newton, L. Sweeney, and B. Malin. Preserving privacy by de-identifying face images. *TKDE*, 2005.

[20] OpenCV Wiki. http://opencv.willowgarage.com/wiki/.

[21] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - A System for Secure Face Identification. In *S&P*, 2010.

[22] F. Samaria and A. Harter. Parameterisation of a stochastic model for human face identification. In *Applications of Computer Vision*, 1994.

[23] A. Senior, S. Pankanti, A. Hampapur, L. Brown, Y. Tian, and A. Ekin. Blinkering Surveillance: Enabling Video Privacy through Computer Vision. *IBM Research Report*, 2003.

[24] Google Maps Street View - Privacy. http://maps.google.com/help/maps/streetview/privacy.html.

[25] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *NDSS*, 2013.

[26] M. Turk and A. Pentland. Eigenfaces for recognition. In *CVPR*, 1991.