Enabling Fine-Grained Permissions for Augmented Reality Applications With Recognizers

Suman Jana[1], David Molnar[2], Alexander Moshchuk[2], Alan Dunn[1], Benjamin Livshits[2], Helen J. Wang[2], and Eyal Ofek[2]

[1]University of Texas at Austin
[2]Microsoft Research

Abstract

Augmented reality (AR) applications sense the environment, then render virtual objects on human senses. Examples include smartphone applications that annotate storefronts with reviews and XBox Kinect games that show "avatars" mimicking human movements. No current OS has special support for such applications. As a result, permissions for AR applications are necessarily *coarse-grained*: applications must ask for access to raw sensor feeds, such as video and audio. These raw feeds expose significant additional information beyond what applications need, including sensitive information such as the user's location, face, or surroundings.

Instead of exposing raw sensor data to applications directly, we introduce a new OS abstraction: the *recognizer*. A recognizer takes raw sensor data as input and exposes higher-level objects, such as a skeleton or a face, to applications. We propose a *fine-grained* permission system where applications request permissions at the granularity of recognizer objects. We analyze 87 shipping AR applications and find that a set of four *core recognizers* covers almost all current apps. We also introduce *privacy goggles*, a visualization of sensitive data exposed to an application. Surveys of 962 people establish a clear "privacy ordering" over recognizers and demonstrate that privacy goggles are effective at communicating application capabilities. We build a prototype on Windows that exposes nine recognizers to applications, including the Kinect skeleton tracker. Our prototype incurs negligible overhead for single applications, while improving performance of concurrent applications and enabling secure offloading of heavyweight recognizer computation.

1 Introduction

An *augmented reality* (AR) application takes natural user interactions (such as gestures, voice, and eye gaze) as input and overlays digital content on top of the real world seen, heard, and experienced by the user. For example, on mobile phones, augmented reality "browsers" such as Layar and Junaio allow users to look through the phone and see annotations about a magazine article or a storefront. Furniture applications on the iPad allow users to preview what a couch would look like in the context of a real room before buying [17]. The Xbox Kinect has sold over 19 million units and allows application developers to overlay avatars on top of a user's pose, creating new kinds of games and natural user interfaces. Microsoft has released a Windows SDK for Kinect and helped incubate multiple startup companies delivering AR experiences on the PC. Even heads-up displays, previously restricted to academic and limited military/industrial use, are set to reach consumers with Google Glass [25].

Today's AR applications are monolithic. The application itself performs sensing, rendering, and user input interpretation (e.g., for gestures), aided by user-space libraries, such as the Kinect SDK, OpenCV [6, 12], or cloud object recognition services, such as Lambda Labs or IQ Engines. Because today's OSes are built without AR applications in mind, they offer only *coarse-grained* access to sensor streams, such as video or audio data. This raises a privacy challenge: it is difficult to build applications that follow the principle of *least privilege*, having access to only the information they need and no more. Today's systems also do not have any AR-specific permissions, relying instead on careful prepublication vetting of applications [5].

Motivating Example. Figure 1 illustrates the problem with coarse-grained abstractions in today's

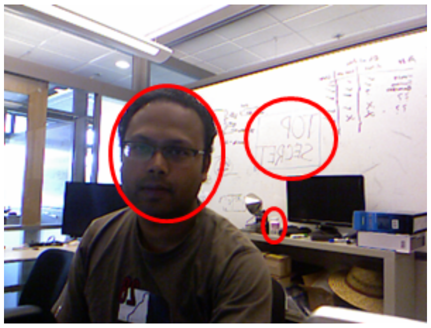

Figure 1: Giving raw sensor data to applications can compromise user privacy. This video frame captured from a Kinect contains the user's face, private whiteboard drawings, and a bottle of medicine.

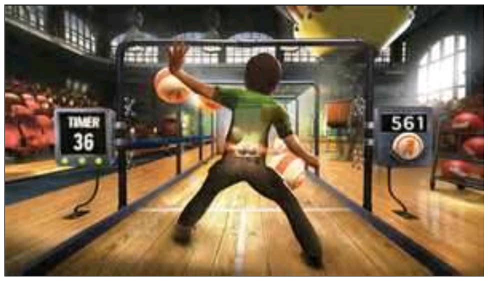

Figure 2: AR applications often need only specific objects rather than the entire sensor streams. The "Kinect Adventures!" game only needs body position to render an avatar and simulate game physics.

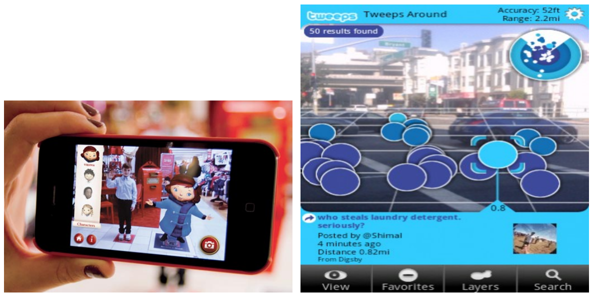

Figure 3: Two examples of mobile AR applications that only need specific objects in a sensor stream. On the left, Macy's Believe-O-Magic only needs the location in the frame of a special marker, on top of which it renders a cartoon character. On the right, Layar only needs to know the GPS location and compass position to show geo-tagged tweets.

Application	Objects recognized
Your Shape 2012	skeleton, person texture
Dance Central 3	skeleton, person texture
Nike+Kinect	skeleton, person texture
Just Dance 4	skeleton, video clip
NBA 2K13	voice commands
Xbox Dashboard	pointer, voice commands
Layar	GPS "points of interest"
Red Bull Racing	Red Bull Cans
Macy's Believe-O-Magic	Macy's store display

Figure 4: Sample AR applications and the objects they recognize. Kinect apps are above the line, mobile below.

AR applications. The figure shows a video frame captured from a camera. Today, applications must ask for raw camera access if they want to do video-based AR, which means the application will see all sensitive information in the frame. In this frame, that information includes the user's face, (private) drawings on the whiteboard, and a bottle of medicine with a label that reveals a medical condition.

An application, however, may not need *any* of this sensitive information to do its job. For example, Figure 2 shows a screenshot from the "Kinect Adventures!" game that ships with the Microsoft Xbox Kinect. First, the game estimates the body position of the player from the video and depth stream of the Kinect. Next, the game overlays an avatar on top of the player's body position. Finally the game simulates interaction between the avatar and a virtual world, including a ball that bounces back and forth to hit blocks. To do its job, the game needs *only* body position, and not any other information from the video and depth stream.

Kinect is just one example of an AR system; this principle of AR applications benefiting from "least privilege" is more general. We show two mobile phone examples in Figure 3. On the left, the Macy's Believe-O-Magic application shows a view of a child standing next to a holiday-themed cartoon character. While the application today must ask for raw video access, which includes the face of the child and of all bystanders, the only information the application needs is the location of a special marker to enable rendering the cartoon in the correct place. On the right, Layar is an "AR browser" for mobile phones, here showing a visualization of where recent tweets have originated near the user. Again, Layar must ask for raw video and location access, but in fact it needs to only know the GPS position of the tweet relative to the user.

Beyond these examples, Figure 4 shows the top 5 Amazon best-selling Kinect-enabled applications for the Xbox 360, along with the Xbox Dashboard and representative AR apps on mobile phones. For each application, as well as the Xbox Dashboard, we enumerate the objects recognized; in Section 5 we carry out a similar analysis for all shipping Xbox Kinect applications. *None* of these applications need continuous access to raw video and depth data, but no current OS allows a user to restrict access at finer granularity.

The Recognizer Abstraction. To address this problem, we introduce a new least-privilege OS abstraction called a *recognizer.* A recognizer takes as input a sensor stream and creates events when objects are recognized. These events contain information about the recognized object, such as its position in the video frame, but not the raw sensor information. By making access to recognizer-exposed objects a first-class permission in an operating system, we enable least privilege for AR applications. We assume a fixed set of system-provided recognizers in this work. This is justified by our analysis of over 87 shipping applications, which shows a set of four "core recognizers" is sufficient for the vast majority of such applications (Section 5).

Supporting recognizers in the OS incurs several benefits. Besides enabling least privilege, recognizers lead to a *performance improvement,* as heavyweight object recognition can be shared among multiple applications. We show how an OS can compose recognizers in a *dataflow graph*, which enables precise reasoning about which recognizers should be run, depending on the set of running applications. Finally, we show how making dataflow explicit allows us to prune spurious permission requests. These benefits extend beyond AR applications and to any set of applications that must interpret higher-level objects from raw sensor data, such as building monitoring, stored video analysis, and health monitoring.

Challenges. We faced several challenges designing our recognizer-based AR platform. First, other fine-grained permission systems, such as Android, have been shown to be difficult to interpret for users [11]. To address this problem, we introduce *privacy goggles*: an "application's-eye view" of the world that shows users which recognizers are available to an application. Users see a video representation of sensitive data that will be shown to the application (Figure 9). This, in turn, lays the foundation for informed permission granting or permission revocation. Our surveys of 462 people show that privacy goggles are effective at communicating capabilities to users.

Another challenge concerned *recognizer errors.* For example, an application may have permission for a skeleton recognizer. If that recognizer mistakenly finds a skeleton in a frame, the application may obtain information even though there is no person present. This information leakage violates a user's expectations, even though the application sees only a higher-level object such as the skeleton.

We address recognizer errors with a new OS component, *recognizer error correction.* We evaluate three approaches: *blurring*, *frame subtraction*, and *recognizer combination.* The first two manipulate raw sensor data to reduce false positives in a recognizer-independent way. The last reduces false positives by using context information available to the OS from its use of multiple recognizers that could not be available to any individual recognizer author. We show that our techniques reduce false positives across a set of seven recognizers implemented in the OpenCV library [12].

Our final challenge concerned recognizers that require heavyweight object recognition algorithms which may run poorly or not at all on performance-constrained mobile devices [23, 21]. We thus build and evaluate support for *offloading* of particularly heavyweight recognizers to a remote machine.

We have implemented a prototype of our system on Windows, using the Kinect for Windows SDK. Our system includes nine recognizers, including face detection, skeleton detection, and a "plane recognizer" built on top of KinectFusion [23].

Contributions. We make the following contributions:

- We introduce a new OS abstraction, the *recognizer*, which captures the core object recognition capabilities of AR applications. Our novel fine-grained permission system for recognizers enables least privilege for AR applications. We show that all shipping Kinect applications would benefit from least privilege. Based on surveys of 500 people, we determine a privacy ordering on common recognizers.
- We introduce a novel visualization of sensitive data provided to AR applications, which we call *privacy goggles.* Privacy goggles let users inspect sensitive information flowing to an application, to aid in permission granting, inspection, and revocation. Our surveys of 462 people show that privacy goggles are effective at communicating capabilities to users.
- We recognize the problem of granting permissions in the presence of object recognition errors and propose techniques to mitigate it.
- We demonstrate that raising the level of abstraction to the "recognizer" enables the OS to offer services such as *offloading* and *cross-application recognizer sharing* that improve performance. Our implementation has negligible

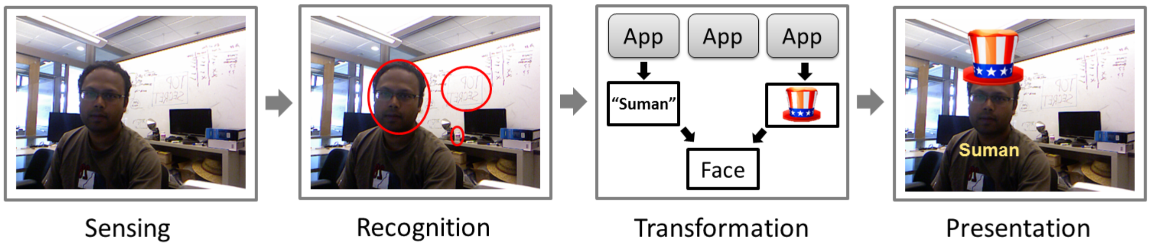

Figure 5: AR application pipeline: (1) reading raw data from hardware, (2) parsing raw data into recognized objects, (3) manipulating these objects to add augmentations to the scene, and (4) resolving conflicts and rendering.

overhead for single applications, yet greatly increases performance for concurrent applications and allows the OS to offload heavyweight recognizer computation.

In the rest of the paper, Section 2 provides background on AR, Section 3 discusses our recognizer abstraction, and Section 4 describes our implementation. Section 5 evaluates privacy goggles, recognizers required for shipping AR applications, recognizer error correction, and performance of our prototype. Sections 6 and 7 present related and future work, and Section 8 concludes.

2 AR Overview

We characterize AR applications using a pipeline shown in Figure 5. First, the *sensing stage* acquires raw video, audio, and other sensor data from platform hardware. In the figure, we show an RGB video frame as an example. The frame was captured from a Kinect, which also exposes a depth stream and a high-quality microphone.

Next, the *recognition stage* applies object recognition algorithms to the raw sensor data. For example, voice recognition may run on audio to look for specific keywords spoken, or a skeleton detector may estimate the presence and pose of a human in the video. As new advances in computer vision and machine learning make it possible to reliably recognize different objects, the resulting algorithms can be added to this stage. The code performing object recognition is similar to drivers in traditional operating systems: code running with high privilege maintains an abstraction between "bare sensing" and applications. Just as with devices in traditional OSes, an OS with support for AR could multiplex applications across multiple object recognition components; we will describe a new OS abstraction that enables this in the next section. In the figure, a face and two areas of text are recognized, one on the whiteboard and another on a bottle of medicine. The output of the recognition stage is a set of *software objects* that "mirror" recognized real-world objects.

In the *transformation stage*, applications consume the recognized objects and add virtual objects of their own. Finally, the *presentation* stage creates the final view for the user, taking as input all current software objects and the current state of the world. This stage must resolve any remaining logical conflicts, as well as check that desired placement of objects is feasible. Today, this rendering is done using standard OS abstractions, such as DirectX or OpenGL.

3 The Recognizer OS Abstraction

We propose a new OS abstraction called a *recognizer*. A recognizer is an OS component that takes a sensor stream, such as video or audio, and "recognizes" objects in the sensor stream. For example, Figure 6 shows a recognizer that wraps face detection logic. This recognizer takes a raw RGB image and outputs a face object if a face is present. The recognizer abstraction lets us capture that most AR applications operate on specific entities with high-level semantics, such as the face or the skeleton. To enable least privilege, the OS exposes higher level entities through recognizers.

Recognizers create *events* when objects are recognized. A recognizer event contains structured data that encodes information about the objects. Each recognizer declares a public type for this structured data that is available to applications. Applications register callbacks with the OS that fire for events from a particular recognizer; the callbacks accept arguments of the specified type. For example, the recognizer in Figure 6 declares that it will return a list of points corresponding to facial features, plus an RGB texture for the face itself. A callback for an application receives the points and texture in its arguments, but not the rest of the raw RGB frame.

The recognizer is the unit of permission granting. Every time an application attempts to register a callback with the OS for a specific recognizer, the application must be authorized by the user. Different

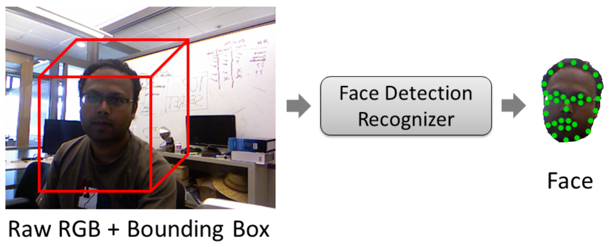

Figure 6: Example of a recognizer for face detection. The input is a feed of raw RGB video plus a region within that video. The recognizer outputs an event if a face is recognized in the region. Applications register callbacks that fire on the event and are called with a list of points outlining the face plus an RGB texture, but not the rest of the video frame.

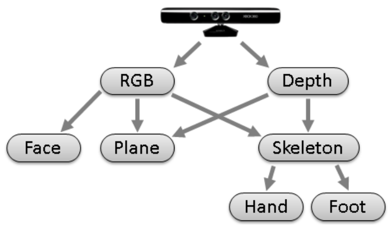

Figure 7: A sample directed acyclic graph of recognizers. Arrows denote how recognizers subscribe to events from other recognizers.

applications can, depending on the user's authorization, have access to different recognizers.This gives us a *fine-grained* permission mechanism.

Users can restrict applications to only "see" a subset of the raw data stream. For example, Figure 6 shows a bounding box in the raw RGB frame that can be associated with a specific application. If a face happened to be present outside this bounding box, that application would not see the resulting event. Such regions are useful to (1) prevent an application from seeing sensitive information in the environment, and (2) improve efficiency and accuracy of recognizers (e.g., by skipping a region that generates false positives). This bounding box works for sensors where the data is spatial, such as RGB, depth, or skeleton feeds. Other cutoffs would work for other sensors, such as filtering audio to a certain frequency range to ensure voice data is not leaked while other sounds are kept.

Recognizers can also *subscribe* to events from other recognizers, just like applications. The OS includes recognizers for raw sensor streams, such as RGB frames from a camera. Because subscribing to events is an explicit call to the OS, the OS can construct a *dataflow graph* showing how raw sensor streams are progressively refined into objects. Figure 7 shows an example. Having explicit data flow

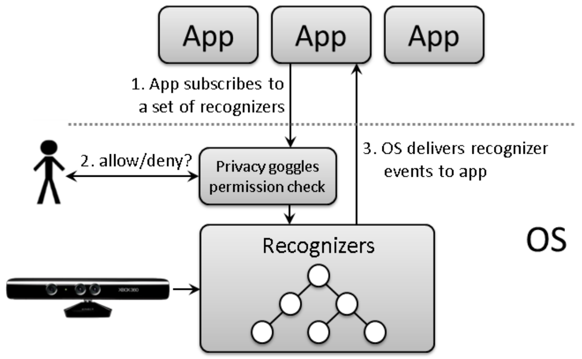
App
App
App
1. App subscribes to a set of recognizers
2. allow/deny?
Privacy goggles permission check
3. OS delivers recognizer events to app
Recognizers
OS

Figure 8: Recognizer-based OS architecture. Applications request subscriptions to sets of recognizers, which the OS then confirms with the user using privacy goggles (Figure 9). Once the user grants permission, the OS delivers recognizer events to subscribed applications.

helps the OS with both security and performance, as we describe below.

Architecture and Threat Model: Figure 8 shows the core architecture of an OS with multiple applications and multiple recognizers. "Root" recognizers acquire raw input from sensors such as the Kinect, then raise events that are consumed by other recognizers. An application may request a subscription for a set of recognizers. The OS confirms this request with the user using our "Privacy Goggles" visualization (Section 3.3). If the user agrees to the request, the OS then delivers events from appropriate recognizers to the application. While our implementation and example focuses on the Kinect, our architecture applies to all forms of object recognition across different platforms such as mobile phones.

The applications are not trusted, while the OS, recognizer implementations, and hardware are trusted. This is similar to the threat model in today's mobile devices. Third-party recognizer implementations are out of scope of this paper, but we describe in Section 7 key new challenges they raise.

3.1 Security Benefits

The recognizer abstraction has two key security benefits:

Least privilege: Applications can be given access only to the recognizers they need, instead of to raw sensor streams. Before recognizers, OSes could expose permissions only at a coarse granularity. As we will see in Section 5, a small set of recognizers is sufficient to cover most shipping AR applications.

Reducing permission requests: If an application requests access to the skeleton and hand recognizers from the DAG shown in Figure 7, a user only needs to grant access to the skeleton recognizer.

More generally, the recognizer DAG allows us to find such dependencies efficiently. This helps with warning fatigue, which is one of the major problems with existing permission systems [11].

3.2 Performance Benefits

Besides the security benefits described above, recognizer DAGs also allow us to achieve significant performance gains.

Sharing recognizer output: Most computer vision algorithms used in recognizers are computationally intensive. Since concurrently running AR applications may access the same recognizers, our recognizer DAG allows us to run such shared recognizers only once and send the output to all subscribed applications. Our experiments show that this results in significant performance gains for concurrent applications.

On-demand invocation: The recognizer DAG allows us to find all recognizers being accessed by currently active applications at all times. We can then prevent scheduling inactive recognizers.

Concurrent execution: The recognizer DAG also allows us to find true data dependencies between the recognizers. We leverage this to schedule independent recognizers in multiple threads/cores and thus minimize inter-thread/core communication.

Offloading: Some recognizers require special-purpose hardware such as a powerful GPU that may not be available in mobile devices. These recognizers must be outsourced to a remote server. For example, the real-time 3D model generation of KinectFusion [23] requires a high-end nVidia desktop graphics card, such as a GeForce GTX 680. Therefore, if we want to use a commodity tablet with a Kinect attached to scan objects and create models, we must run the recognizer on a remote machine. While applications could implement offloading themselves, adding offloading support to the OS preserves least privilege. For example, the OS can offload KinectFusion without giving applications access to raw RGB and depth inputs, which would be required if an application were to offload it manually.

3.3 Privacy Goggles

We introduce *privacy goggles*, an "application-eye view" of the world for running applications. For example, if the application has access to a skeleton recognizer, a stick figure in the "privacy goggles view" mirrors the movements of any person in view of the system, as shown in Figure 9. A trusted visualization method for each recognizer communicates the capabilities of applications that have access to this recognizer. If an application requests access to more than one recognizer, the OS will compose the appropriate visualizations. In Section 5 we survey 462 people to demonstrate that privacy goggles do effectively communicate capabilities for "core recognizers" derived from analyzing shipping AR applications. Privacy goggles are complementary to existing permission widgets, such as those of Howell and Schechter [16], which allow users to understand how apps perceive them in real time.

Permission Granting and Revocation. Privacy goggles lay a foundation for permission granting, inspection, and revocation experiences. For example, we can generalize existing install-time manifests to use privacy goggles visualizations. At installation time, a short prepared video could play showing a "raw" data stream side by side with the privacy goggles view. The user can then decide to allow access to all, some, or none of the recognizers. We are currently evaluating this approach. Because manifest-based systems have known problems with user attention [11], we are also exploring how access-control gadgets might interact with privacy goggles [27].

A major difference between privacy goggles and existing permission granting systems like Android manifests is the visual representation of the sensitive data. The visual representation helps users to make informed decisions about granting and revoking an application's access to different recognizers. Traditional systems do not need this representation because they ask for permissions about well-understood low-level hardware, such as the camera and microphone. Because we are fine-grained and must consider higher-level semantics, we need privacy goggles to show the impact of allowing applications access to specific recognizers.

After installation, privacy goggles are a natural way to inspect sensitive data exposed to applications. The user can trigger a "privacy goggles control panel" to zero in on a particular application or view a composite for all applications at once. From the control panel, a user can then turn off an application's access to a recognizer or even uninstall the application.

3.4 Handling Recognizer Errors

Because our permission system depends on recognizer outputs, we have a new challenge: *recognizer errors*. Object recognition algorithms inside recognizers have both false positives and false negatives. A false negative means that applications will not "see" an object in the world, impacting functionality.

False negatives, however, do not concern privacy.

A false positive, on the other hand, means that an application will see more information than was intended. In some cases the damage will be limited, because the recognizer will return information that is not sensitive. For example, a false positive from a recognizer for hand positions is unlikely to be a problem. In others, false positives could leak portions of raw RGB frames or other more sensitive data.

To address recognizer errors, we introduce a new OS component for *recognizer error correction.* While recognizers themselves implement various techniques to decrease errors, in our setting false positives are damaging, while false negatives are less important. Therefore, we are willing to tolerate more false negatives and fewer false positives than a recognizer developer who is not concerned with basing permission decisions on a recognizer's output.

For recognizer error correction, we first considered two techniques: *blurring* and *frame subtraction*, both of which are well-known graphics techniques that can be applied in a *recognizer-independent* way. We apply these techniques to recognizer inputs to reduce potential false positives, accepting that they may raise false negatives. We discuss the results and show data in Section 5.

In addition, the OS has information not available to an individual recognizer developer: results from *other recognizers* in the same system on the same environment. Recognizer error correction can therefore employ *recognizer combination* to reduce false positives. For example, if a depth camera is available, the OS can use the depth camera to modify the input to a face detection recognizer. By blanking out all pixels past a certain depth, the OS can ensure a face recognizer focuses only on possible faces near the system. While combination does require knowing something about what a recognizer does, it is independent of the internals of the recognizer implementation. For another example, the OS can combine a skeleton recognizer and a face recognizer to release a face image only if there is also a skeleton with its head in the appropriate place.

3.5 Adding New Recognizers.

Today's AR platforms ship with a small fixed set of recognizers. Applications that want capabilities outside that set need to both innovate on object recognition and on app experience, which is rare. As the platforms mature, we expect additional recognizers to appear. The main incremental costs for new recognizers are 1) coming up with a privacy goggles visualization, 2) measuring the effectiveness of this visualization at informing users (and re-designing if not effective), and 3) defining relationships with existing recognizers to support recognizer error correction. For example, a new "eye recognizer" would have the invariant that every eye detected should be on a head detected by the skeleton recognizer. Third-party recognizers raise additional security issues outside the scope of this paper; we discuss them briefly in Section 7.

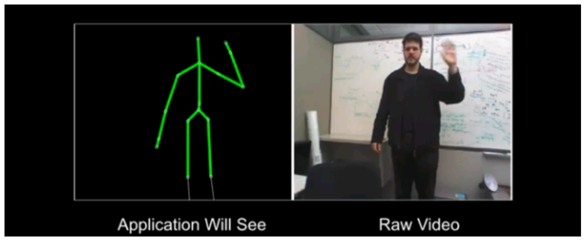

Figure 9: Example of "privacy goggles." The user sees the "application-eye view" for a skeleton recognizer.

API	Purpose
`init`	Register
`destruct`	Clean up
`event_generate`	Notify apps of recognized objects
`visualize`	Render recognized objects
`filter`	Restrict domain for recognition
`cache_compare`	Compare to previous inputs

Figure 10: The APIs implemented by each recognizer. The first four are required, while `filter` and `cache_compare` are optional.

4 Implementation

We have built a prototype implementation of our architecture. Our prototype consists of a *multiplexer*, which plays the role of an OS "kernel", and *ARLib*, a library used by AR applications to communicate to the multiplexer. Our system uses the Kinect RGB and depth cameras for its sensor inputs.

Multiplexer. The multiplexer handles access to the sensors and also contains implementations of all recognizers in the system. Our applications no longer have direct access to Kinect sensor data and must instead interact with the multiplexer and retrieve this data from recognizers. The multiplexer supports simultaneous connections from multiple applications. To simplify implementation, we built the multiplexer as a user-space program in Windows that links against the Kinect for Windows SDK.

The multiplexer registers each recognizer using a static, well-known name. Applications use these names to request access to one or more recogniz-

```
var client = new MultiplexerClient();
client.Connect();
client.OnFace += new FaceEventCallback(ProcessFace);
...
public void ProcessFace(FTPoint[] points)
{
    if (points.Length > 0) {
        DrawFace(points);
    } else {
        RemoveFace();
    }
}
```

Figure 12: Code used by a sample C# application to connect to the multiplexer, subscribe to events from the face recognizer, and use those events to update its face visualization.

ers. When the multiplexer receives such an access request, it asks the user whether or not permission should be granted using privacy goggles (Section 3.3). If the user grants permission, the multiplexer will forward future recognizer events, such as face mesh points from a face recognizer, to the application.

The multiplexer interacts with recognizers via an API shown in Figure 10. All recognizers must implement the first four API calls. The multiplexer calls `init` to initialize a recognizer and `destruct` to let a recognizer release its resources. In our current implementation, the multiplexer calls the `event_generate` function of each recognizer in a loop, providing prerequisite recognizer inputs as parameters, to check if any new objects have been recognized. If so, the recognizer will return data that the multiplexer will then package in an event data structure and pass to all subscribed applications. We plan to implement a more efficient interrupt-driven multiplexer in the future.

The next two API calls are optional. The `filter` call allows the multiplexer to tell the recognizer that only a specific subset of the raw inputs should be used for recognition. For example, only a sub-rectangle of the video frame should be considered for a face detector. Finally, `cache_compare` is a recognizer-specific comparator function that takes two sets of recognizer inputs and determines whether they are considered equal. The multiplexer uses this comparator to implement per-recognizer caching. For example, the multiplexer may pass the previous and current RGB frames to the `cache_compare` function of the face recognizer and potentially avoid a recomputation of the face model if the two frames have not sufficiently changed.

Our multiplexer and recognizers consisted of about 3,000 lines of C++ code. We wrote a total of nine recognizers, which we summarize in Figure 11.

Application support. Applications targeting our multiplexer run in separate Windows processes. Each application links against the *ARLib* library we have built. ARLib communicates with the multiplexer over local sockets and handles marshaling and unmarshaling of recognizer event data. By calling ARLib functions, an application can request access to specific recognizers and register callbacks to handle recognizer events. ARLib provides two kinds of interfaces: a low-level interface for applications written in C++ and higher-level wrappers for .NET applications written in C# or other managed languages. ARLib consists of about 500 lines of C++ code and 400 lines of C# code.

Sample code in Figure 12 shows a part of a test application we wrote that detects faces and draws pictures on the screen which follow face movements. The application connects to the multiplexer and subscribes to face recognizer events. In our implementation, these events contain approximately 100 points corresponding to different parts of the face, or 0 points if a face is not present. The application handles these events in the `ProcessFace` callback by checking if a face is present and calling a separate function (not shown) that updates the display.

In addition to face visualization, we ported a few other sample applications bundled with the Kinect SDK to our system. These included a skeleton visualizer and raw RGB and depth visualizers. We found the porting effort to be modest, aided in part by the fact that we modeled our event data formats on existing Kinect SDK APIs. In each case, we only changed a handful of lines dealing with event subscription. We additionally wrote two applications from scratch: a 500-line C++ application that translates hand gestures into mouse cursor movements, and a 300-line C# application that uses face recognition to annotate people with their names. Overall, we found our multiplexer interface simple and intuitive to use for building AR applications.

5 Evaluation

We first evaluate how recognizers are used by an analysis of 87 shipping AR applications and users' mental models of AR applications. A survey of 462 respondents shows that users expect AR applications to have limited access to raw data. Furthermore, no shipping application needs continuous RGB access, and in fact a set of four recognizers is sufficient for almost all applications. For these "core" recognizers, we design privacy goggles visualizations and evaluate how well users understand them. Next, we look at how the OS can mitigate recognizer er-

Recognizer	Input dependencies	Output
RGB	*Kinect*	RGB camera frames
Depth	*Kinect*	Depth camera frames
Skeleton	*Kinect*	Computed skeleton model(s)
Hand	Skeleton	Hand positions
FaceDetect	RGB	2D face models for faces in current view
PersonTexture	Depth, Skeleton	Depth "cutout" of a person
Plane	RGB, Depth	3D polygon coordinates constructed with KinectFusion (see Section 5.3)
FaceRecognize	RGB, FaceDetect	Name of person in current view (see Section 5.3)
CameraMotion	*Kinect*	Camera movements detected using an accelerometer/gyro

Figure 11: The nine recognizers implemented by our multiplexer. A "Kinect" input dependency means that the recognizer obtains data directly from the Kinect rather than other recognizers.

rors once an application has access to recognizers. Finally, we show that our abstraction enables performance improvements, making this a rare case when improved privacy leads to improved performance.

5.1 Recognizers

Core Recognizers. We analyzed 87 AR applications on the Xbox Kinect platform, including all applications sold on Amazon.com. We focused on Kinect because it is widely adopted and sits in a user's home. For each application, we manually reviewed their functionality, either through reading reviews or by using the application. From this, we extracted "recognizers" that would be sufficient to support the application's functionality.

Figure 13 shows the results. Four *core recognizers* are sufficient to support around 89% of shipping AR applications. The set consists of skeleton tracking, hand position, *person texture*, and keyword voice commands. Person texture reveals a portion of RGB video around a person detected through skeleton tracking, but with the image blurred or otherwise transformed to hide all details. Fitness applications, in particular, use person texture when instructing the user on proper form.

After the core set, there is a "long tail" of seven recognizers. For example, the Alvin and the Chipmunks game uses voice modulation to "Alvin-ize" the player's voice, and NBA Baller Beats actually tracks the location of a basketball to check that the player dribbles in time to music. *None* of the applications in our set, however, require continuous access to RGB data. Instead, applications take a short video or photo of the player so that she can share how silly she looks with friends; this could be handled via user-driven access control [27]. Only 3 applications require audio access beyond voice command triggers. There is plenty of room to improve privacy with least privilege enabled by the recognizer abstraction.

Recognizer	% Apps
Skeleton	94.3%
Person Texture (PT)	25.3%
Voice Commands (VC)	3.44%
Hand Position (HP)	5.74%
Video Clip	3.4%
Picture Snap	1.1%
Voice Intensity	1.1%
Voice Modulation	1.1%
Speaker Recognition	1.1%
Sound Recognition	1.1%
Basketball Tracking	1.1%
Skeleton+PT+VC	82.75%
Skeleton+PT+VC+HP	89.65%

Figure 13: Analysis of all recognizers used by 87 shipping Xbox applications. For each recognizer, we show what percentage of apps use that recognizer (and possibly others). We also show two sets of recognizers, and for each set, the percentage of apps that use recognizers in this set and no others. A set of four recognizers covers 89.65% of all applications. No application needs continuous raw RGB access, and only 3 need audio access beyond voice commands.

Privacy Expectations for Applications. To learn users' mental models of AR application capabilities, we showed 462 survey respondents a video of a Kinect "foot piano" application in action: the Kinect tracks foot positions and plays music. We then asked about the capabilities of the application. Figure 17(A) shows the results. Over 86% of all users responded that the application could see the foot positions, while a much smaller number believed this application had other capabilities. Overall, users expect applications will not see the entire raw sensor stream.

Privacy Goggles for Core Recognizers. As we discussed in Section 3, every recognizer must implement a visualization method to enable the privacy goggles view. The OS uses these visualizations to display to the user what information is obtained by each application. We developed privacy goggles visualizations for three of the four core recognizers: skeleton, hand position, and person texture. While voice commands are also a core recognizer, You have found an app that you want to install. Right before installing, the video below plays. On the right, a sample "raw" video. On the left, a view of what the app will see if it is installed. Which of the following can the app do?

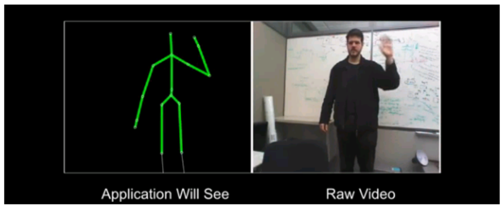

Figure 14: Example survey question for privacy goggles. An embedded warning video shows two views: the raw video on the right, and what the application will see on the left. Survey respondents watched the warning video, then answered questions about what the app could or could not do after installation. Out of 152 respondents, 80% correctly identified that the app could see body position, and 47% correctly determined the app could see hand positions.

Consider the two pictures below. Which picture contains "more sensitive" information?

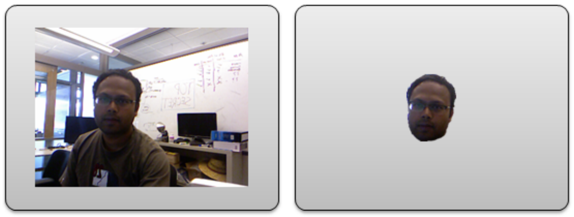

Figure 15: Example survey on relative sensitivity. Respondents indicated which picture is more sensitive: the "raw" RGB video frame or an image showing only the output of a face detector. Out of 50 respondents, 86% indicated the raw image was more sensitive.

we decided to focus first on the visual recognizers and leave visualization of voice commands for future work.

Privacy Attitudes for Core Recognizers. We then conducted surveys to measure the relative sensitivity of the information released by the core recognizers. We also added the "face detector" recognizer, because intuitively the face is private information, and a "Raw" video recognizer that represents giving all information to the application. For each pair of recognizers, we showed a visualization from the same underlying video frame, then asked the participant to state which picture was "more sensitive" and why. Figure 15 shows an example comparing raw RGB and face detector recognizers.

For each pair of recognizers, we asked 50 people to rate which picture contained information that was "more sensitive." Figure 16 shows the results. In total we had 500 survey respondents, all from the United States. As expected, respondents find that the raw RGB frame is more sensitive than any other

Recognizers		**Left more**	**95%**
Left	**Right**	**sensitive**	**CI**
Raw	Face	86%	± 9.6%
Raw	Skeleton	78%	± 11.48%
Raw	Texture	88%	± 9.01%
Raw	Hand	88%	± 9.01%
Texture	Skeleton	82%	± 10.65%
Texture	Face	35%	± 13.22%
Texture	Hand	84%	± 10.16%
Skeleton	Face	24%	± 11.84%
Skeleton	Hand	84%	± 10.16%
Hand	Face	22%	± 11.48%

Figure 16: Results from relative sensitivity surveys. Users were shown two pictures, one from each recognizer, here shown as the "left" and the "right" recognizer. The table reports which picture respondents thought contained "more sensitive" information and the 95% confidence interval. For example, in the first line, 86% of people thought that the view from the "Raw" RGB recognizer was more sensitive than the view from a face detector, with a 95% confidence interval of ± 9.6%.

recognizer. Based on the responses, we can order recognizers from "most sensitive" to "least sensitive", as follows: Raw, Face, Person Texture, Skeleton, and finally the least sensitive is Hand Position.

Effectiveness of Privacy Goggles. Finally, we evaluated whether our "privacy goggles" visualizations successfully communicate the capabilities of applications. We created three surveys, one for each of the skeleton, person texture, and hand recognizers. We had at least 150 respondents to each survey, with a total of 462 respondents. Our surveys are inspired by Felt *et al.*'s Android permission "quiz." [11]

We showed a short video clip of the privacy goggles visualization for the target recognizer. Figure 14 shows an example for the skeleton recognizer. The right half shows the raw RGB video of a person writing on a whiteboard and handling a small ceramic cat figurine. The left half shows the "application-eye view" showing the detected skeleton. We then asked users what they believed the capabilities of the application would be if installed. Figure 17 shows the results, with a check mark next to correct answers. We see that a large number of respondents (over 80%) picked the correct result and relatively few picked incorrect results. This shows that privacy goggles are effective at communicating application capabilities to the user.

Respondent Demographics. Our survey participants were recruited from the U.S. through uSample [30], a professional survey service, via the Instant.ly web site. We did not specify any restrictions on demographics to recruit. As reported by uSample, participants are 66% female and 33%

A. Foot Piano (462 respondents)	
See my body position	76 (16%)
See my foot positions ✓	400 (86%)
See what I look like	28 (6%)
See the entire video	52 (11%)
Learn my heart rate	21 (4%)
None of the above	20 (4%)
I don't know	20 (4%)

B. Skeleton (152 respondents)	
See what I look like	17 (11%)
See my body position ✓	122 (80%)
See my location	24 (16%)
Read the contents of the whiteboard	14 (9%)
Send premium SMS messages on my behalf	4 (3%)
Track the position of my hands ✓	71 (47%)
None of the above	4 (3%)
I don't know	1 (1%)

C. Person Texture (156 respondents)	
See what I look like	36 (23%)
See my body position ✓	137 (88%)
See my location	25 (16%)
See the ceramic cat	19 (12%)
Read the contents of the whiteboard	5 (3%)
Send premium SMS messages on my behalf	0 (0%)
Track the position of my hands ✓	60 (38%)
None of the above	2 (1%)
I don't know	5 (3%)

D. Hand Position (154 respondents)	
See what I look like	17 (11%)
See my body position	32 (21%)
See my location	14 (9%)
See the ceramic cat	12 (8%)
Read the contents of the whiteboard	7 (5%)
Send premium SMS messages on my behalf	2 (1%)
Track the position of my hands ✓	125 (81%)
None of the above	3 (2%)
I don't know	4 (3%)

Figure 17: Results from privacy goggles effectiveness surveys. For each of our three core recognizers, we first asked respondents to answer questions about the capabilities of a Kinect "foot piano" application based on a short video of the application in use (A). We next showed a privacy goggles "permission warning video" and asked questions about what the application could do if installed (B-D).

male, with 10.2% in the 0–22 age range, 12.9% 22–26, 21.2% 26–34, 16.8% 34–42, 13.5% 42–50, 15.1% 50–60, 8.1% 60–70, and 1.8% 70 or older.

Human Ethics Statement. Our experiments include surveys of anonymous human participants. Our institution does not have an Institutional Review Board (IRB), but it does have a dedicated team whose focus is privacy and human protection. This team has pre-approved survey participant vendors to ensure that they have privacy policies which protect participants. We followed the guidelines of this team in choosing our survey vendor. We also discussed our surveys with a member of the team to ensure that our questions did not ask for personally identifiable information, that they were not overly intrusive, and that no other issues were present.

5.2 Noisy Permissions

While privacy goggles are effective at communicating what an app should and should not see to the user, the recognizers we use can have false positives. These could leak information to applications. We first evaluated a representative set of recognizers on well-known vision data sets to quantify the problem. Next, we evaluated OS-level mitigations for false positives.

Recognizer Accuracy. We picked three well-known data sets for our evaluations: (1) a Berkeley dataset consisting of pictures of objects, (2) an INRIA dataset containing pictures of a talking head, and (3) a set of pictures of a face turning toward the camera and then away. We then evaluated baseline false positive and false negative rates for seven object recognition algorithms contained in the widely adopted OpenCV library. All seven had false positives on at least one of the data sets.

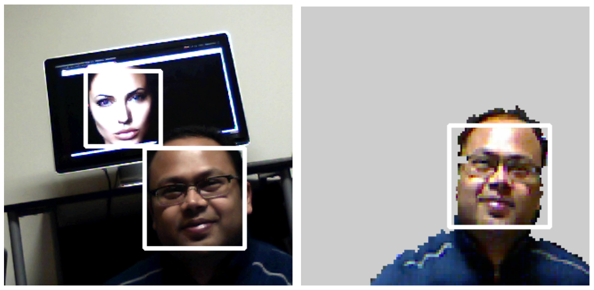

Figure 19: Recognizer combination in action. The left figure shows results of running a face detector on a raw RGB video frame. Two faces are detected, but only one belongs to a real person. On the right, face detection is run after combining RGB and depth. Only the real person is detected.

Input Massaging. We then implemented *pre-permission blurring*, in which frames are put through a blurring process using a box filter before being passed to the face detection algorithm. We used a 12×12 box filter. We also used *frame subtraction* as a heuristic to suppress recognizer false positives. In frame subtraction, when a recognizer detects an object with a bounding box b in a frame F_1 that it did not detect in the previous frame F_0, we compute the difference $Crop(F_1, b) - Crop(F_0, b)$ and check the number of pixels that have a difference. If this num-

Recognizer	Data Set	False Positive	False Negative	BlurFP	BlurFN	SubFP	SubFN
Face	Objects	10.6%	0%	6%	0%	9.6%	0%
Face	Talking Head	0.2%	0%	0%	0%	0%	0%
Face	Turning Face	19.1%	16.1%	15%	16.1%	17.64 %	16.1%
FullBody	Objects	14.8%	0%	3.5%	0%	9.6%	0%
FullBody	Talking Head	0.2 %	0%	0%	0%	0%	0%
FullBody	Turning Face	24.6%	0%	22.7 %	0%	20%	0%
LowBody	Objects	19.5%	0%	4.6%	0%	17.9%	0%
LowBody	Talking Head	6.2%	0%	0.3%	0%	0%	0%
LowBody	Turning Face	33%	0%	25%	0%	28.3%	0%
UpperBody	Objects	41%	0%	10%	0%	38.1%	0%
UpperBody	Talking Head	5.3%	0%	0.1%	0%	0.2%	0%
UpperBody	Turning Face	86%	0%	0%	0%	19.9%	0%
Eye	Object	35%	0%	83%	0%	32 %	0%
Eye	Talking Head	64 %	0 %	100 %	0%	30%	2%
Eye	Turning Face	23 %	5%	100%	0%	9%	10%
Nose	Object	17.8%	0%	57%	0%	17.1 %	0%
Nose	Talking Head	90 %	0%	86%	0%	90%	0%
Nose	Turning Face	24.5 %	0%	43%	7%	24%	0%
Mouth	Object	61%	0%	75%	0%	59 %	0%
Mouth	Talking Head	100 %	0%	75%	0%	100%	0%
Mouth	Turning Face	75 %	0%	82%	0%	74%	0%

Figure 18: False positive and false negative rates for OpenCV recognizers on common data sets. False positives are important because they could leak unintended information to an application. We also show the effect of blurring and frame subtraction. For blurring we used a 12x12 box filter.

ber does not exceed a threshold, we ignore the detected object as a false positive.

For three out of our seven recognizers, blurring decreases false positives with no effect on false negatives, with a maximum reduction for our lower body recognizer from 19.5% false positives to 4.6% false positives. For the remaining recognizers, false positives decrease but false negatives also increase. Frame subtraction decreases false positives for six out of seven recognizers and has no effect on the seventh, with no impact on false negatives. This is in line with our goals, because false positives are more damaging to privacy than false negatives. The full results are in Figure 18.

Recognizer Combination. Finally, we implemented *recognizer combination*, in which the OS can take advantage of the fact that multiple recognizers are available. Specifically, we combined the OpenCV face detector with the Kinect depth sensor. We chose the OpenCV face detector because its developers could depend only on the presence of RGB video data. We ran an experiment that first acquires an RGB and depth frame, then blanks out all pixels with depth data that is further away than a threshold. Next, we fed the resulting frame to the face detector. An example result is shown in Figure 19. On the left, the original frame shows a false positive detected behind the real person. On the right, recognizer combination successfully avoids the false positive.

Recognizers	Kinect SDK	Our framework
RGB Video	29.87 fps	30.02 fps
Skeleton	29.59 fps	28.65 fps
Face	28.24 fps	28.00 fps

Figure 20: Frame rates for a single application using the Kinect SDK vs. using recognizers from our system. Our system incurs negligible overhead.

5.3 Performance

In our performance evaluation, we (1) measure the overhead of using our system compared to using the Kinect SDK directly, (2) quantify the benefits of recognizer sharing for multiple concurrent applications, and (3) evaluate the benefit of recognizer offloading.

Overhead over Kinect SDK. Compared to directly using the Kinect SDK, an application that uses our multiplexer will face extra overhead due to recognizer event processing in the multiplexer as well as data marshaling and transfer over local sockets. To quantify this overhead, we wrote two identically functioning applications to obtain and display a raw 640x480 RGB video feed, a skeleton model, and points from a face model. The first application used the Kinect SDK APIs directly, while the second

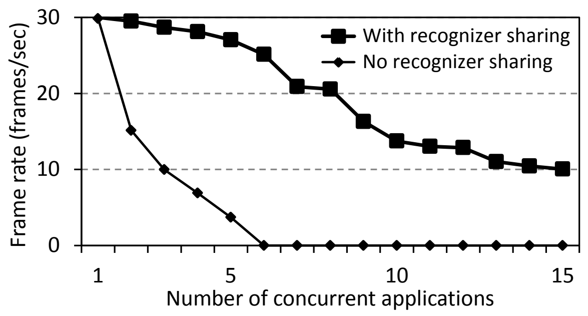

Figure 21: Effect of sharing a concurrent RGB video stream between applications. Our framework enables 25 frames per second or higher for up to six applications, while without sharing the frame rate drops.

used our multiplexer with RGB, skeleton, and face detection recognizers.

Figure 20 shows the frame rates when running these two applications on a desktop HP xw8600 machine with a 4-core Core i5 processor and 4 GB of RAM. We see that, fortunately, our current prototype incurs negligible overhead over the Kinect SDK when used by a single application.

Recognizer Sharing. Next, we ran multiple concurrent copies of the two applications above to evaluate the benefits of recognizer sharing as well as the scalability of our prototype. Since the Kinect SDK does not permit concurrent applications, we wrote a simple wrapper for simulating that functionality, i.e., allowing multiple applications as if they were linking to independent copies of the Kinect SDK.

Figure 21 shows the average frame rate for multiple concurrent applications using the RGB recognizer. We see that without recognizer sharing, frame rates quickly stall as the number of concurrent applications increases, becoming unusable beyond five applications. In contrast, our approach maintains at least 25 frames per second up to six concurrent applications and degrades gracefully thereafter. We experienced similar recognizer sharing benefits for skeleton and face recognizers.

While currently shipping AR platforms do not yet support multiple concurrent applications, the above experiment demonstrates that our system is ready to *efficiently* embrace such support. Indeed, we believe this to be the future of AR platforms. Mobile phone "AR Browsers" such as Layar already expose APIs for third-party developers, with over 5,000 applications written for Layar alone [20]. Users will benefit from running these applications concurrently; for example, looking at a store front, one application may show reviews of the store, while another shows information about its online catalog, and yet a third application attaches a name to the face of someone walking by.

	Throughput (frames/sec)		
Recognizer	Tablet	Offloaded	Server
Plane detection	0	4.17	4.46
Face recognition	2.04	2.73	2.84

Figure 22: Frames processed per second when running recognizers (1) locally on a client tablet, (2) offloaded to the server and shipping results back to the tablet, and (3) locally on the server.

Recognizer Offloading. We evaluated offloading of two resource-intensive recognizers: plane and face recognition. The plane recognizer reconstructs planes in the current scene using KinectFusion, which computes 3D models from Kinect depth data [23]. The face recognizer uses the Microsoft Face SDK [21] to identify the name of the person in the scene using a small database of known faces.

We implemented offloading across two devices linked by an 802.11g wireless network. For face recognition, the client sends RGB bitmaps of the current scene to the server as often as possible; the client additionally includes the depth bitmap for the plane recognizer.

Our client device was a Samsung Series 7 tablet running Windows 8 Pro 64-bit with a 2-core Core i5 processor and 4 GB of RAM, hooked up to a Kinect. Our server device was a desktop HP Z800 machine running Windows 8 Pro 64-bit with two 4-core Xeon E5530 processors, 48 GB of RAM, and an Nvidia GTX 680 GPU.

The first two columns of Figure 22 show throughputs experienced by the client when running recognizers locally and when offloading them to the server. The plane recognizer requires a high-end Nvidia GPU, which prevented it from running on our client at all; we report this as zero frames per second. With offloading, however, the client is able to detect planes 4.2 times per second. For face recognition, the client processed 2.73 frames per second when offloading, a 34% improvement in response time compared to running face recognition locally. In addition, when run locally, face recognition placed heavy CPU load on the client, completely consuming one of its two cores. With offloading, the client's CPU consumption dropped to 15% required to send bitmaps, saving battery and freeing resources for processing other recognizers. Note that our setup allows the offloading server to service multiple clients in parallel. For example, the server was able to handle eight concurrent face recognition clients before saturating.

We also considered the overhead of our offloading mechanism by plugging a Kinect into our server and running the recognizer framework directly on it. Column 3 of Figure 22 shows these results. We see that offloading with the Kinect on the client is only 4–7% slower than running the Kinect on the server, meaning that the offloading overhead of transferring bitmaps and recognition results is reasonable.

6 Related Work

Augmented Reality. Azuma surveyed augmented reality, defining it as real-time registration of 3-D overlays on the real world [1], later broadening it to include audio and other senses [2]. We take a broader view and also consider systems that take input from the world. Qualcomm now has an SDK for augmented reality that includes features such as marker-based tracking for mobile phones [26]. Previous work by our group has laid out a case for adding OS support for augmented reality applications and highlighted key challenges [7].

Common shipping object recognition algorithms include skeleton detection [29], face and headpose detection [31, 21], and speech recognition [22]. More recently, Poh *et al.* showed that heart rate can be extracted from RGB video [24]. Our recognizer graph and simple API allow quickly adding new recognizers to our system.

Sensor Privacy. There are several parts to sensor privacy: access control on sensors, sensor data usage control once an application obtains access to sensor data, and access visualization; we discuss related work for each.

Access control can take the form of user permissions. iOS's permission system is to prompt a user at the first time of the sensor access (such as a map application first accessing GPS). Android and latest Windows OSes use manifests at application installation time to inform the user of sensor usage among other things; the installation proceeds only if the user permits the application to permanently access all the requested permissions. These existing permission systems are either disruptive or ask users' permissions out-of-context. They are not least-privilege; permanent access is often granted unnecessarily. Felt et al [11] has shown that most people ignore manifests, and the few who do read manifests do not understand them. To address these issues, access control gadgets (ACGs) [27] were introduced to be trusted UI elements for sensors, which are embeddable by applications; users' authentic actions on an ACG (e.g., a camera trusted UI) grants the embedding application permission to access the represented sensor. In this paper, we argue that even the ACG style of permission granting is too coarse-grained for augmented reality systems because most AR applications only require specific objects rather than the entire RGB streams (Section 5.1).

Another form of access control is to reduce the sensitivity of private data (e.g., GPS coordinates) available to applications. MockDroid [3] and AppFence [14] allow using fake sensor data. Krumm [19] surveys methods of reducing sensitive information conveyed by location readings. Differential privacy [9] uses well-known methods for computing the amount of noise to add to give strong guarantees against an adversary's ability to learn about any specific individual. Similarly, we proposed modifying sensor inputs to recognizers in specific ways to reduce false positives that could result in privacy leaks. Darkly [18] transforms output from computer vision algorithms (such as contours, moments, or recognized objects) to blur the identity of the output. Darkly can be applied to the output of our recognizers.

Once an application obtains access to sensors, information flow control approaches can be used to control or monitor an application's usage of the sensitive data as in TaintDroid [10] and AppFence [14].

In access visualization, sensor-access widgets [15] were proposed to reside within an application's display with an animation to show sensor data being collected by the application. Darkly [18] also gives a visualization on its transforms (see above). Our privacy goggles apply similar ideas to the AR environment, allowing a user to visualize an application's eye view of the user's world.

Abstractions for Privacy. Our notion of taking raw sensor data and providing the higher-level abstraction of recognizers is similar to CondOS [4]'s notion of Contextual Data Units. However, they neither choose a set of concrete Contextual Data Units that are suitable for a wide variety of real-world applications nor address privacy concerns that arise from applications having access to Contextual Data Unit values. Koi [13] provides a location matching abstraction to replace raw GPS coordinates in applications. The approach in Koi is limited to location data and may require significant work to integrate into real applications, while our recognizers cover many types of sensor data and were specifically chosen to match application needs.

7 Future Work

Further Recognizer Visualization. The recognizers we evaluated had straightforward visualizations, such as the Kinect skeleton. As we noted, some recognizers, such as voice commands, do not have obvious visualizations. Other recognizers might extract features from raw video or audio for use by a variety of object recognition algorithms, but not in themselves have an easily understood semantics, such as a fast Fourier transform of audio. One key challenge here is to design visualizations for privacy goggles that clearly communicate to users the impact of allowing application access to the recognizer. For example, with voice commands we might try showing a video with sound where detected words are highlighted with subtitles. A second key challenge is characterizing the privacy impact of algorithmic transforms on raw data, especially in the case of computer vision features that have not been considered from a privacy perspective.

Third-Party Recognizers. All the recognizers in this paper are assumed trusted. To enable new experiences, we would like to support extension of the platform with third-party recognizers. Supporting third-party recognizers raises challenges, including permissions for recognizers as well as sandboxing untrusted GPU code without sacrificing performance. We have developed recognizers in a domain-specific language that enables precise analysis [8]. Dealing with such challenges is intriguing future work, similar in spirit to research on third-party driver isolation in an OS. For example, we might require such recognizers to go through a vetting program and then have their code signed, similar to drivers in Windows or applications on mobile phone platforms.

Sensing Applications. Besides traditional AR applications, other applications employ rich sensing but do not necessarily render on human senses. For example, robots today use the Kinect sensor for navigating environment, and video conferencing can use the "person texture" recognizer we describe. One of our colleagues has also suggested that video conferencing can benefit from a depth-limited camera [28]. These applications may also benefit from recognizers.

Bystander Privacy. Our focus is on protecting a user's privacy against untrusted applications. Mobile AR systems such as Google Glass, however, have already raised significant discussion of *bystander privacy* — the ability of people around the user to opt out of recording and object recognition. Our architecture allows explicitly identifying all applications that might have access to bystander information, but it does not tell us when and how to stop sending recognizer events to applications. Making the system aware of these issues is important future work.

8 Conclusions

We introduced a new abstraction, the *recognizer*, for operating systems to support augmented reality applications. Recognizers allow applications to raise the level of abstraction from raw sensor data, such as audio and video streams, to ask for access to specific recognized objects. This enables applications to act with the least privilege needed. Our analysis of existing applications shows that all of them would benefit from least privilege enabled by an OS with support for recognizers. We then introduced a "privacy goggles" visualization for recognizers to communicate the impact of allowing access to users. Our surveys establish a clear privacy ordering on core recognizers, show that users expect AR apps to have limited capabilities, and demonstrate privacy goggles are effective at communicating capabilities of apps that access recognizers. We built a prototype on top of the Kinect for Windows SDK. Our implementation has negligible overhead for single applications, enables secure OS-level offloading of heavyweight recognizer computation, and improves performance for concurrent applications. In short, the recognizer abstraction improves privacy *and* performance for AR applications, laying the groundwork for future OS support of rich sensing and AR application rendering.

9 Acknowledgements

We thank Janice Tsai, our Privacy Manager, for reviewing our survey. We thank Doug Burger, Loris D'Antoni, Yoshi Kohno, Franziska Roesner, Stuart Schechter, Margus Veanes, and John Vilk for helpful discussions and review of drafts. Stuart Schechter suggested the idea of a depth-limited camera for teleconferencing scenarios. This work was carried out while the first and fourth author were interning at Microsoft Research.

References

[1] R. T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.

[2] R. T. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computer Graphics and Applications*, 21(6):34–47, 2001.

[3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[4] D. Chu, A. Kansal, J. Liu, and F. Zhao. Mobile apps: It's time to move up to condOS. May

2011. http://research.microsoft.com/apps/pubs/default.aspx?id=147238.

[5] M. Corporation. Kinect for xbox 360 privacy considerations, 2012. http://www.microsoft.com/privacy/technologies/kinect.aspx.

[6] M. Corporation. Kinect for Windows SDK, 2013. http://www.microsoft.com/en-us/kinectforwindows/.

[7] 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. J. Wang. Operating system support for augmented reality applications. In *Hot Topics in Operating Systems (HotOS)*, 2013.

[8] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. FAST: A transducer-based language for tree manipulation, 2012. MSR Technical Report 2012-123 http://research.microsoft.com/apps/pubs/default.aspx?id=179252.

[9] C. Dwork. The differential privacy frontier. In *6th Theory of Cryptography Conference (TCC)*, 2009.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Conference on Operating System Design and Implementation*, 2010.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[12] W. Garage. OpenCV, 2013. http://opencv.org/.

[13] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *NSDI*, 2012.

[14] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Conference on Computer and Communications Security*, 2011.

[15] J. Howell and S. Schechter. What You See is What They Get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy, IEEE*, 2010.

[16] J. Howell and S. Schecter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy Workshop*, 2010.

[17] E. Hutchings. Augmented reality lets shoppers see how new furniture would look at home, 2012. http://www.psfk.com/2012/05/augmented-reality-furniture-app.html.

[18] S. Jana, A. Narayanan, and V. Shmatikov. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *IEEE Symposium on Security and Privacy*, 2013.

[19] J. Krumm. A survey of computational location privacy. *Personal Ubiquitous Computing*, 13(6):391–399, Aug 2009.

[20] Layar. Layar catalogue, 2013. http://www.layar.com/layers.

[21] Microsoft Research Face SDK Beta. http://research.microsoft.com/en-us/projects/facesdk/.

[22] Microsoft Speech Platform. http://msdn.microsoft.com/en-us/library/hh361572(v=office.14).aspx.

[23] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *10th IEEE International Symposium on Mixed and Augmented Reality*, 2011.

[24] M. Poh, D. MacDuff, and R. Picard. Advancements in non-contact, multiparameter physiological measurements using a webcam. *IEEE Trans Biomed Engineering*, 58(1):7–11, 2011.

[25] Project Glass. https://plus.google.com/+projectglass/posts.

[26] Qualcomm. Augmented Reality SDK, 2011. http://www.qualcomm.com/products_services/augmented_reality.html.

[27] 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 *IEEE Symposium on Security and Privacy*, 2011.

[28] S. Schecter. Depth-limited camera for skype - personal communication, 2012.

[29] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from a single depth image. In *Computer Vision and Pattern Recognition*, June 2011.

[30] uSample. Instant.ly survey creator, 2013. http://instant.ly.

[31] P. Viola and M. Jones. Robust Real-time Object Detection. In *International Journal of Computer Vision*, 2001.