Copyright
by
Suman Jana
2014

The Dissertation Committee for Suman Jana
certifies that this is the approved version of the following dissertation:

Security and Privacy in Perceptual Computing

Committee:

Vitaly Shmatikov, Supervisor

Lorenzo Alvisi

Dan Boneh

Úlfar Erlingsson

Emmett Witchel

Security and Privacy in Perceptual Computing

by

Suman Jana, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2014

Dedicated to my wife Baishakhi and my parents.

Acknowledgments

First, I would like to thank my family, friends, and mentors without whose continuous support I would have never been able to finish my PhD.

I am grateful to my advisor, Vitaly Shmatikov, for steering me towards the right direction in each step of the PhD program. He taught me how to distill interesting research problems out of real world issues, critically evaluate my own research, and how to present the results both in written and verbal forms. He patiently listened to all of my ideas (even the ones that were not distilled enough to be expressed as research problems), explained their merits and demerits to me, and finally let me decide whether I want to pursue them or not. Also, besides research, I have found his advice and guidance about other issues in life to be invaluable as well. He is a great mentor, collaborator, and friend without whom I would not have been able to complete this journey.

I would like to thank my thesis committee members for their thoughtful feedback that helped me to improve this thesis significantly. Part of the work that resulted in this thesis was done while I was doing internships at Microsoft Research and Google. I am thankful to my mentors at Microsoft Research: David Molnar, Alexander Moshchuk, Benjamin Livshits, Helen J. Wang, and Eyal Ofek for their guidance, support, and help. David Molnar introduced me to the idea of security and privacy issues in augmented reality systems. I would also like to thank my

fellow Microsoft Research intern Alan Dunn for his help with this work. I am grateful to Úlfar Erlingsson, my internship mentor at Google, for his support and feedback. Special thanks go to Karen Lees and Iulia Ion who helped in designing and executing several experiments as part of this work.

I am indebted to Arvind Narayanan for his critical input on the design and implementation of DARKLY. I am thankful to Richard McPherson for collaborating with me in the evaluation of the security and privacy properties of the augmented reality browsers. I am also grateful to Scott Saponas, Stuart Schechter, Loris D'Antoni, Margus Veanes, and Ryan Calo for helpful discussions during the early stages of this work and to Piyush Khandelwal for helping me evaluate DARKLY on the Segway RMP-50 robot.

I thank my fellow UT students Alan Dunn, Martin Georgiev, Owen Hofmann, Amir Houmansadr, Michael Lee, Don Porter, Sooel Son, and Srinath Setty for lively discussions about research that I enjoyed a lot. I thank Lindy Aleshire and Lydia Griffth to help me take care of all the necessary paper work for this thesis.

I am very grateful to my family for their unconditional love and support through out this journey. My mother taught me the value of perseverance. I got my critical reasoning and thinking skills from my father. I am also thankful to the rest of my family for their continuous support.

Finally, I am grateful beyond words to my wife Baishakhi Ray for critical feedback on this work and for her constant encouragement and support.

Security and Privacy in Perceptual Computing

Publication No. ____________________

Suman Jana, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Vitaly Shmatikov

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 dissertation’s main thesis is that perceptual applications present several new classes of security and privacy risks to both their users and the bystanders. Existing perceptual platforms are often completely inadequate for mitigating these risks. For example, we show that the augmented reality browsers, a class of popular perceptual platforms, contain numerous inherent security and privacy flaws.

The key insight of this dissertation is that perceptual platforms can provide stronger security and privacy guarantees by controlling the interfaces they expose to the applications. We explore three different approaches that perceptual platforms can use to minimize the risks of perceptual computing: (i) redesigning the perceptual platform interfaces to provide a fine-grained permission system that allows least-privileged application development; (ii) leveraging existing perceptual

interfaces to enforce access control on perceptual data, apply algorithmic privacy transforms to reduce the amount of sensitive content sent to the applications, and enable the users to audit/control the amount of perceptual data that reaches each application; and (iii) monitoring the applications' usage of perceptual interfaces to find anomalous high-risk cases.

To demonstrate the efficacy of our approaches, first, we build a prototype perceptual platform that supports fine-grained privileges by redesigning the perceptual interfaces. We show that such a platform not only allows creation of least-privileged perceptual applications but also can improve performance by minimizing the overheads of executing multiple concurrent applications. Next, we build DARKLY, a security and privacy-aware perceptual platform that leverages existing perceptual interfaces to deploy several different security and privacy protection mechanisms: access control, algorithmic privacy transforms, and user audit. We find that DARKLY can run most existing perceptual applications with minimal changes while still providing strong security and privacy protection. Finally, We introduce peer group analysis, a new technique that detects anomalous high-risk perceptual interface usages by creating peer groups with software providing similar functionality and comparing each application's perceptual interface usages against those of its peers. We demonstrate that such peer groups can be created by leveraging information already available in software markets like textual descriptions and categories of applications, list of related applications, etc. Such automated detection of high-risk applications is essential for creating a safer perceptual ecosystem as it helps the users in identifying and installing safer applications with any desired

functionality and encourages the application developers to follow the principle of least privilege.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiv**

List of Figures **xv**

Chapter 1. Introduction **1**

1.1 Different stages of perceptual computation 4
1.2 Perceptual computing ecosystem 7
1.3 Architecture of today's perceptual applications 10
1.4 Threat model 12
1.5 Security & privacy risks of perceptual computation 14
1.5.1 Security risks 15
1.5.2 Privacy risks 16
1.6 Towards secure and private perceptual computation 18
1.6.1 Basic principles 18
1.6.2 Our contributions 22

Chapter 2. Overview of related work **26**

2.1 Security and privacy issues in perceptual applications 26
2.2 Privacy-preserving perceptual computing 27
2.3 Sensor security and privacy 28
2.4 Protecting bystander's privacy from perceptual applications 31

Chapter 3. Security and privacy flaws in existing perceptual platforms 33
3.1 Introduction . . . 33
3.2 Related work . . . 38
3.3 AR services . . . 40
3.3.1 Functional requirements . . . 41
3.3.2 Components of AR services . . . 42
3.3.3 Specific AR browsers . . . 44
3.4 Threat model . . . 45
3.5 Access to native resources . . . 47
3.5.1 Doing it wrong . . . 48
3.5.2 Risks . . . 49
3.5.3 How to do it right . . . 51
3.6 Support for non-HTML AR content . . . 52
3.6.1 Doing it wrong . . . 54
3.6.2 Risks . . . 56
3.6.3 How to do it right . . . 58
3.7 Image-triggered code execution . . . 60
3.7.1 Doing it wrong . . . 60
3.7.2 Risks . . . 61
3.7.3 How to do it right . . . 65
3.8 Outsourced image processing . . . 65
3.8.1 Doing it wrong . . . 66
3.8.2 Risks . . . 66
3.8.3 How to do it right . . . 67
3.9 Visual composition . . . 68
3.9.1 Doing it wrong . . . 68
3.9.2 Risks . . . 69
3.9.3 How to do it right . . . 69
3.10 Indirect retrieval of AR content . . . 70
3.10.1 Doing it wrong . . . 71
3.10.2 Risks . . . 72
3.10.3 How to do it right . . . 73
3.11 Conclusions . . . 74

Chapter 4. Redesigning perceptual interfaces with recognizers **75**
4.1 Introduction 75
4.2 The recognizer abstraction 81
4.2.1 Security benefits 85
4.2.2 Performance benefits 85
4.2.3 Privacy goggles 87
4.2.4 Handling recognizer errors 88
4.2.5 Adding new recognizers. 90
4.3 Implementation 90
4.4 Evaluation 95
4.4.1 Recognizers 95
4.4.2 Noisy permissions 100
4.4.3 Performance 104
4.5 Related work 108
4.6 Future work 111
4.7 Conclusions 112

Chapter 5. DARKLY: retrofitting privacy protection in existing perceptual interfaces **114**
5.1 Introduction 114
5.2 Threat model and design of DARKLY 115
5.3 Privacy risks of perceptual applications 119
5.4 Structure of perceptual applications 122
5.5 Design principles of DARKLY 124
5.6 Implementation 126
5.6.1 OpenCV 127
5.6.2 Opaque references 129
5.6.3 Interposition 130
5.6.4 Privacy transforms 132
5.6.5 Trusted services 133
5.6.6 Support for application-provided code 136
5.7 Privacy transforms 141
5.7.1 Sketching 142

5.7.2 Generalization 144
5.8 DARKLY console 149
5.9 Evaluation 150
5.10 Related work 156
5.11 Conclusions 158

Chapter 6. Peer group analysis: automated detection of applications violating least privilege 160

6.1 Introduction 160
6.2 Securing online software markets with peer group analysis 166
6.3 Do users expect security privileges to be tied to software functionality? 169
6.3.1 User reports of privilege abuse 170
6.3.2 User study results 172
6.4 Estimating unexpectedness using peer group analysis 174
6.4.1 Identifying peer groups 174
6.4.2 Estimating security-relevant behavior 181
6.4.3 Estimating unexpectedness 182
6.5 Experimental setup 182
6.6 Evaluation 184
6.6.1 Effects of peer group parameters 185
6.6.2 Effectiveness of peer group analysis 187
6.7 Related work 190
6.8 Conclusion 192

Chapter 7. Conclusions 194

Bibliography 196

Vita 210

List of Tables

4.1 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. 103

5.1 Transforms used for each DARKLY declassifier. 133

5.2 Benchmark OpenCV applications. 152

5.3 Evaluation of DARKLY on OpenCV applications. 153

6.1 Variation of unexpectedness across the search results returned for different search queries. 188

List of Figures

1.1 A sample posture improvement application that tracks the user's posture over time using video frames from the camera and provides suggestions about better posture. 5

1.2 Different stages of perceptual computation. Note that the last two stages are optional. 7

1.3 Perceptual computing ecosystem. 8

1.4 Architecture of today's perceptual applications. The gray boxes indicate trusted components of the system. 11

1.5 Sample video frame captured from a Kinect containing multiple pieces of sensitive information: the face of the author, drawings on the whiteboard, and a bottle of medicine with the label showing. . . 11

3.1 A Layar-based mobile app [33]. 35

3.2 Basic structure of AR services. 35

3.3 Architecture of a typical AR service. 42

3.4 Architecture of an AR browser. 43

3.5 Examples of a Junaio channel showing a 3D model placed over the Junaio logo. 45

3.6 A Layar channel running on top of a scanned magazine page. The AR objects are circled. Clicking any of the colors below the 3D watch model changes its color. The user can also add the watch to his or her shopping cart. 46

3.7 Junaio's visual stack. AR objects are on top of the camera feed, the transparent overlay on top of the objects. If an object is clicked, a pop-up appears at the very top. 52

3.8 Layar's visual stack. AR objects, which can include HTML pages, are overlaid on the camera feed. 54

3.9 Cross-site scripting (XSS) in Junaio 56

3.10 Universal XSS vulnerability in Junaio. 57

3.11 Both codes launch the same channel, but Layar fails to parse the code on the right and does not show the URL. 61

3.12 Depending on the angle, each poster nondeterministically launches either its own channel, or the channel associated with the other poster. 63
3.13 Different combinations of the Junaio mascot and the QR code launch different channels. 64
3.14 An image sent by the Layar browser over HTTP so that the Layar server can recognize content triggers. Note the accidentally captured credit card. 66
3.15 Clickjacking in Layar. 70
3.16 User authentication in Layar. 71
3.17 Layar cookie stealing attack. 73

4.1 Perceptual 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. 76
4.2 Two examples of mobile perceptual 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. 77
4.3 Sample perceptual applications and the objects they recognize. Kinect apps are above the line, mobile below. 78
4.4 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. 83
4.5 A sample directed acyclic graph of recognizers. Arrows denote how recognizers subscribe to events from other recognizers. 83
4.6 Recognizer-based perceptual platform architecture. Applications request subscriptions to sets of recognizers, which the OS then confirms with the user using privacy goggles (Figure 4.7). Once the user grants permission, the platform delivers recognizer events to subscribed applications. 84
4.7 Example of "privacy goggles." The user sees the "application-eye view" for a skeleton recognizer. 90
4.8 The APIs implemented by each recognizer. The first four are required, while `filter` and `cache_compare` are optional. 91

4.9 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. 93

4.10 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. 94

4.11 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. 97

4.12 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. 98

4.13 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. 99

4.14 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%. 100

4.15 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). 101

4.16 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. 102

4.17 Frame rates for a single application using the Kinect SDK vs. using recognizers from our system. Our system incurs negligible overhead. 105

4.18 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. . 106

4.19 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. 107

5.1 System architecture of DARKLY. 116

5.2 Output of the sketching transform on a female face image at different privacy levels. 137

5.3 Output of the sketching transform on a credit card image at different privacy levels. 138

5.4 Sketching: reduction in information available to the application for images from Figs. 5.2 and 5.3. 144

5.5 Sketching: reduction in average information available to the application for facial images in FERET database (size roughly 220x220). 145

5.6 Face morphing for generalization. The left and right faces belong to the same cluster; the morph “representing” this cluster is in the center. 148

5.7 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. 148

5.8 Motion detector: actual image and the DARKLY console view. Application works correctly with the transformed image. 149

5.9 Ball tracker: actual image and the DARKLY console view. Application works correctly with the transformed image. 149

5.10 Frame rates with and without DARKLY. 154

5.11 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. 155

5.12 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. 159

6.1 The description and related items for the Evernote software in the Chrome Web Store market. 162

6.2 Permission screen shown while installing the Evernote extension from the Chrome Web Store. 163

6.3 User expectations of different security privileges (making a phone call, read SMS messages, access the internet, read the contacts on the phone, take pictures, read calendar entries, find out phone's location, and use the microphone) required by applications with different functionality (social networking, shopping, gaming, messaging). 171

6.4 Word cloud showing the words taken from the 'video download helper' extension's textual description hosted in the Chrome Web Store. The font sizes of the words are proportional to their frequencies. 175

6.5 Word cloud showing top 20 words of a topic detected by LDA that consists mostly of games. The font sizes of the words are proportional to their probability of selection under the topic. 178

6.6 Word cloud showing top 20 words of a topic detected by LDA that consists mostly of 'movies' and 'videos'. The font sizes of the words are proportional to their probability of selection under the topic. 178

6.7 The related software items for the Google Translate extension in the ChromeMarket. 180

6.8 Distribution of the number of related extensions extracted for each extension hosted in Chrome Web Store. 180

6.9 Variability of the percentage of applications with at least one unexpected privileges, for different $relative_frequency_threshold$ choices. 186

6.10 Variability of the percentage of outlying extensions with unexpected privileges, for different peer group sizes. 186

6.11 Variability of the percentage of applications with no unexpected privileges, for different privilege types. 187
6.12 Distribution of the percentages of suspended extensions with unexpectedness score in Chrome Web Store. 189

Chapter 1

Introduction

One of the emerging trends among modern software is increasing support for *perceptual* functionality. Such functionality allows the software to interact with their users by observing them in their physical environment using different sensors like cameras, microphones, etc. We collectively refer to them as *perceptual software*. These software come in many forms: "natural user interface" systems that interact with users via gestures and sounds, image recognition applications such as Google Goggles, monitoring software such as motion detectors and face recognizers, augmented reality applications, "ambient computing" frameworks, a variety of video-chat and tele-presence programs, etc.

The common requirement for executing perceptual software on a system is the availability of high-bandwidth sensors. Most modern devices usually come with multiple such sensors. Widespread availability of sensors allow perceptual applications to execute on a diverse set of hardware platforms including 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. Most of these hardware provide support for running third-party perceptual applications through online software

markets i.e. “app stores”. For example, `robotappstore.com` (“your robots are always up-to-date with the coolest apps”) allows consumers to download and execute thousands of third-party perceptual applications on household robots.

One of the hardest parts of developing these perceptual applications is processing raw sensor data and extracting high-level semantics like detecting the presence of an object. Consider the case of a simple gesture-controlled perceptual application that allows controlling the audio volume using hand gestures. The application developer has to detect the presence of a hand in the input image, recognize the gesture made by the hand, and change the audio volume accordingly. This is a tall order for a regular developer without specific computer vision know-how. To make such tasks easier for third-party developers, several platform owners have designed different perceptual computing platforms like Microsoft Kinect SDK [22], augmented reality browsers [44], and OpenCV [70] that provide API support for extracting high-level information from the raw sensor data. For example, Microsoft Kinect SDK provides library functions for detecting the outline of a human body (i.e., skeleton) to help the application developers in creating gesture-controlled applications. OpenCV provides more than 500 functions for detecting high-level features (e.g., edges, shapes, and contours etc.) from input images. Augmented Reality (AR) browsers provide abstractions for performing tasks like outsourced image matching to enable faster development of AR content using web technologies (i.e., HTML, CSS, and JavaScript). AR is a special class of perceptual computation that, besides processing perceptual input from the sensors, also enhance users’ perception of the surrounding world by blending interactive virtual objects with the

visual representation of actual objects in real time [5, 6]. Perceptual computing platforms have gained significant popularity —the OpenCV library has been downloaded more than 7 million times, AR browsers (e.g., Junaio, Layar, and Wikitude together) have more than 30 million users, and more than 24 million units of Micorsoft Kinect have been sold so far.

However, despite their tremendous potential, untrusted perceptual applications pose serious security and privacy threats to both their users and other people in the vicinity, as these applications can spy on the people by scanning their surroundings. For example, perceptual applications running in one's home or a public area may conduct unauthorized surveillance, intentionally or unintentionally over-collect 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. Even more disturbingly, a programmable robot can move around and focus on specific targets. Some of these security and privacy concerns have already become a reality: many people are uncomfortable with law enforcement agencies conducting large-scale facial recognition [13, 61].

In this dissertation, we systematically analyze the security and privacy risks posed by the emerging trends of perceptual computation and propose solutions to mitigate the threats. By analyzing the design of existing perceptual computing platforms (e.g. AR browsers, Microsoft Kinect SDK, OpenCV) we show that these platforms allow untrusted applications unrestricted access to sensitive perceptual data regardless of their functionality. We also perform a thorough evaluation of

the security and privacy properties of AR browsers, one of the most popular perceptual platforms, and found several serious security vulnerabilities resulting from their ad hoc design. However, modifying these platforms to protect sensitive data is non-trivial as it is difficult to isolate sensitive information that are intermingled with benign perceptual content in the data streams coming out of the sensors. Our key thesis is that perceptual platforms can provide stronger security and privacy guarantees by controlling the interfaces they expose to the applications. This dissertation explores three different approaches for building secure and privacy-aware perceptual computing platforms and implement two different prototypes with complementary techniques for improving security and privacy.

The rest of this chapter is organized as follows. We provide an overview of different stages of perceptual computation in Section 1.1 followed by a description of the perceptual computing ecosystem in Section 1.2. We analyze the structure of today's perceptual applications in Section 1.3. Section 1.4 summarizes the threat model that we consider in this dissertation. We provide a hierarchy of different types of security and privacy risks resulting from perceptual computation in Section 1.5. We conclude the chapter by summarizing the solutions that we explore to mitigate such risks in Section 1.6.

1.1 Different stages of perceptual computation

In this section, we describe four abstract stages of perceptual computation in detail: accessing sensor data, extracting high-level semantic content, modifying parts of the semantic content, and displaying some parts of the modified content

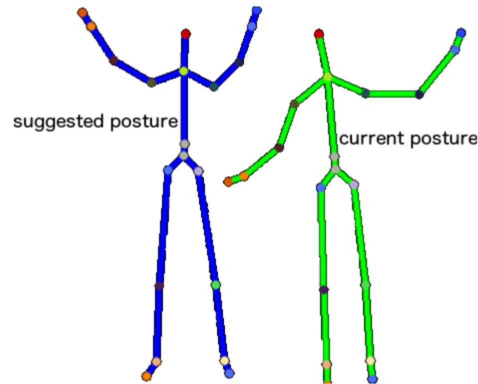


Figure 1.1: A sample posture improvement application that tracks the user's posture over time using video frames from the camera and provides suggestions about better posture.

back to the user. While all perceptual applications implement first two stages, the last two stages are only used by a subset of perceptual software like AR applications. We explain the functionality of each of the stages below. To demonstrate how these abstract stages are used by real applications, we use a simple posture improvement application that takes video frames from the camera as input, shown in Figure 1.1, as a running example throughout this section. The posture improvement application simply monitors the user's posture over time and suggests improvements.

- **Accessing sensor data.** All perceptual applications, by definition, need access to read data from different sensors like camera, microphone etc. Many perceptual applications may access the sensors even when they are executing in the background and the users are not interacting with these applications or any other applications running on the device. For example, the posture

monitoring application (shown in Figure 1.1) may read video frames from the camera even when the user is not using the device in order to provide feedback about the user's posture later.

- **Extracting semantic content from sensor data.** For most practical purposes, perceptual applications must understand the contents of raw sensor data before it can perform any useful task for the user. Therefore, all perceptual applications must extract high-level semantic information from the sensor data. For example, our posture improvement application must detect the presence of humans in the image streams coming out of the camera and estimate their poses.

  Extracting semantic content from the sensor data is usually computationally expensive as they involve complex computer vision or speech recognition algorithms. Perceptual applications running on resource-constrained devices like mobile phones often outsource such computations to remote third-party perceptual services. Several different providers like QualComm Vuforia [19] or Catchoom [20] provide cloud-based web services for locating and recognizing different objects in a given input image.

- **Modifying extracted semantic content.** After extracting high-level content from the sensor data, some perceptual applications, as part of their functionality, also need to manipulate these contents by either adding new objects or modifying/deleting existing objects. For example, the posture improvement application may want to show the suggested posture together with the current

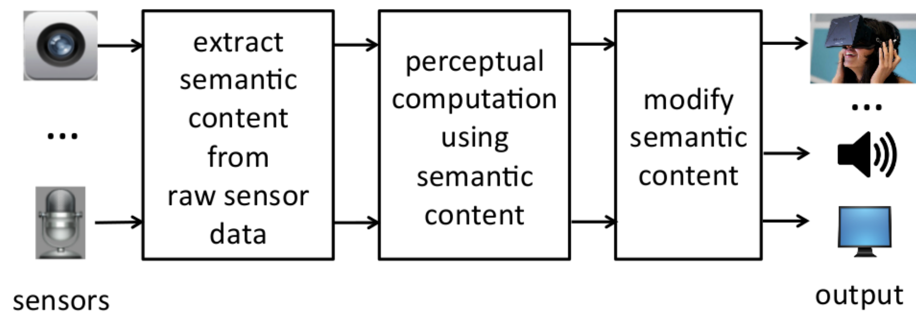


Figure 1.2: Different stages of perceptual computation. Note that the last two stages are optional.

posture of the user as shown in Figure 1.1. This requires the application to add the outline of a human body with the suggested posture along with the extracted posture of the user.

- **Display the modified semantic content.** Finally, the perceptual applications modifying the semantic information extracted from the sensors also need to display the information through different output devices like LCD monitors, VR headsets, etc. Our posture monitor application, in order to be useful, must provide feedback about the user's posture by displaying the suggested posture together with the current posture of the user.

1.2 Perceptual computing ecosystem

The perceptual computing ecosystem consists of several different entities as shown in Figure 1.3. Users install third-party perceptual applications on their devices from online software markets (e.g., Apple app store, Google play store). Once

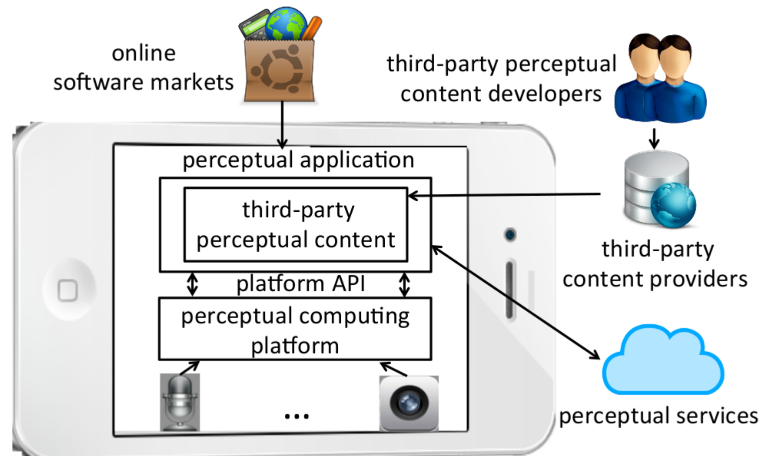


Figure 1.3: Perceptual computing ecosystem.

executed by the users, the perceptual applications interact with the sensors using the API functions provided by the corresponding perceptual platform. Some perceptual applications may fetch and execute third-party perceptual content (e.g., advertisements, web content) hosted in third-party servers as shown in Figure 1.3. Similarly, some perceptual applications may also outsource computationally-intensive tasks like detecting objects in an image to third-party perceptual web services. We describe the major components of this ecosystem in detail below.

Online software markets. Online software markets host a diverse set of software (including perceptual software) created by application developers. The hosted software may either be available free of cost or for a certain developer-determined price. Most of the online software markets (e.g., Apple app store, Google play store) are maintained by the platform owners. The software markets usually provide users with the facilities for performing keyword searches for particular applications,

browsing all available applications, or viewing a list of related applications. Once a user has located the desired application in a market and paid for the application, she can install it on her device directly from the online market.

Perceptual application developers. Perceptual application developers create their applications for different platforms and publish them through the corresponding online software markets. The development process of perceptual applications usually is platform-specific as the APIs for accessing perceptual data vary significantly across different platforms.

Perceptual platform API. Perceptual applications interact with the platforms through the APIs exposed by the platforms. Depending on the platform, the API functions work at different levels of abstraction. For example, OpenCV lets the applications extract high-level features like outline of objects, edges etc. using functions like $cvFindContours$ and $cvCanny$. Microsoft Kinect SDK operates at a higher level of abstraction than OpenCV and let the applications directly detect objects like human faces, skeletons, etc. using API functions like $GetFaceRect$, $NuiSkeletonGetNextFrame$ respectively.

Third-party perceptual content. Most perceptual applications often fetch and execute some third-party content. Such content usually is fetched over the network during application execution . Application developers include third-party content for different reasons ranging from monetization through advertisements to supporting new features using third-party modules/content. For example, an application may include a perceptual advertisement for earning money from the advertisement

providers or incorporate perceptual third-party web content to provide new functionality like displaying user-submitted annotations overlayed on top of different objects.

Third-party content providers. Third-party perceptual content is usually hosted in content servers. These servers may be maintained by someone other than the content developers. For example, advertisements are usually hosted by the advertisement providers like Google, Microsoft etc. The content provider servers must ensure that the perceptual application accessing a particular content is actually authorized to do so. The provider must also ensure that the third-party content do not get modified in transit.

Perceptual services. Perceptual services allow applications running on resource-constrained devices to outsource computation-intensive perceptual tasks like object recognition, facial emotion detection, etc. The perceptual services must ensure that the sensitive perceptual data sent by the applications do not get leaked to an untrusted party during either transmission over the network or processing of the data in remote machines.

1.3 Architecture of today's perceptual applications

Today's perceptual applications get direct access to the raw sensor data and are responsible for extracting semantic content from the data either by themselves or by using APIs provided by perceptual platforms like OpenCV or Microsoft Kinect SDK. Figure 1.4 shows how existing perceptual applications are structured. These

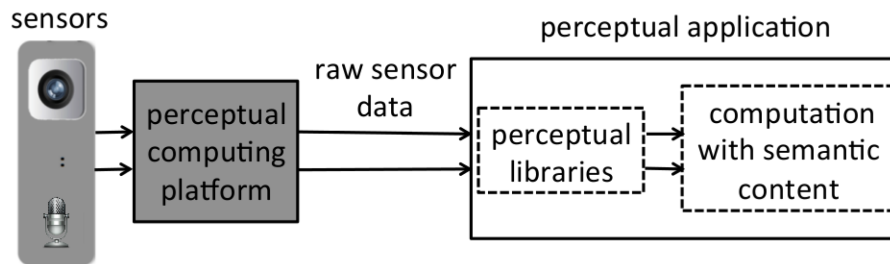


Figure 1.4: Architecture of today's perceptual applications. The gray boxes indicate trusted components of the system.

Figure 1.5: Sample video frame captured from a Kinect containing multiple pieces of sensitive information: the face of the author, drawings on the whiteboard, and a bottle of medicine with the label showing.

applications can only run one at a time, and they receive unrestricted and exclusive access to read the input sensors and perform their computations. However, this approach has two major drawbacks, user privacy and lack of support for concurrent applications, which we discuss below.

First, it is undesirable to give any untrusted application complete access to video and other sensor streams. Consider Figure 1.5, which shows a video frame captured from a Kinect using the Kinect for Windows SDK. In the current architecture of perceptual applications, any application using the SDK has access to the

raw video and depth stream. In this case, that includes the users face, the contents of the whiteboard, and a bottle of medicine. This is a severe privacy risk. Therefore, we must rethink how the perceptual computing platforms interpret and deliver perceptual input to applications while preserving user privacy.

Second, rather than continue running perceptual applications one at a time, it is desirable and compelling to let multiple AR applications from different vendors simultaneously read sensor inputs and perform their computations. Being able to run concurrent tasks is an essential functionality of all advanced computing platforms. Similarly, an multi-application perceptual platform will be more desirable over execution of one application at a time. Existing perceptual platforms like mobile phone AR browsers, such as Layar, have already created rich and diverse APIs for third parties to write applications on top of Layar. Currently, Layar sports over five thousand applications in its Layar Catalogue [58], each of which adds different annotations to the world. However, the lack of support for executing them concurrently often limit the usefulness of such annotations.

1.4 Threat model

We primarily focus on the scenario where the users are executing untrusted third-party perceptual applications on their devices. However, the device running these applications, its operating system, and the hardware of its perceptual sensors are trusted as shown in Figure 1.4. The perceptual applications can be arbitrarily malicious, but they run with user-level privileges and can only access the system, including perceptual sensors, through a trusted API provided by the corresponding

perceptual computing platform. For the applications using third-party cloud-based perceptual services, we assume that the services are honest but may be curious and thus may overcollect user-specific perceptual data. In our model, the attacker may also compromise the network over which the applications communicate with these services. For the perceptual applications that support execution of third-party code like syndicated advertisements, we also assume that the third-party code can be arbitrarily malicious. We summarize the different classes of attackers in perceptual ecosystem in detail below.

Untrusted perceptual applications. We assume that perceptual applications are untrusted and thus either erroneously or maliciously can overcollect user-specific perceptual data and send it over to remote servers. We classify untrusted applications in two different classes: malicious and benign but buggy applications. Malicious applications willingly overcollect sensitive perceptual data and exfiltrate it. By contrast, benign but carelessly designed buggy perceptual applications may leak sensitive data to other applications or remote servers accidentally.

Untrusted perceptual applications are very common in practice as most of the users often install different third-party perceptual applications from online software markets and have little visibility into the applications' inner workings. Although, the market owners often try to detect and remove blatantly malicious applications, they are not known to be very effective against applications surreptitiously collecting privacy-sensitive information about their users.

Curious perceptual services. Many perceptual applications tend to outsource data

and computation intensive parts of perceptual computation to third-party perceptual services as shown in Figure 1.3. In our model, we assume these services to be honest in the sense that they provide their advertised functionality correctly. However, we allow the services to be curious i.e. they may store the perceptual data sent to them by applications and peek into it for identifying and collecting targeted user-specific data.

Malicious third-party code in perceptual applications. As Figure 1.3 shows, perceptual application developers may include third-party contents such as syndicated advertisements in their applications for better monetization. We assume that an attacker can trick any perceptual application that includes such third-party content into incorporating her malicious content, e.g., via ad brokers.

Network attackers. We assume that the network over which the perceptual applications communicate (either with third-party perceptual services or third-party content providers as shown in Figure 1.3) can be controlled by the attacker either through man-in-the-middle attacks or by being on the same network as the victim. A network attacker can listen in on the communications between the perceptual application and the perceptual services or any other remote servers.

1.5 Security & privacy risks of perceptual computation

In this section, we categorize and describe different types of new security and privacy risks that are posed by perceptual computation.

1.5.1 Security risks

- **Lack of access control for perceptual data.** Today's perceptual computing platforms do not support enforcing fine-grained access control over the perceptual data accessed by different applications through the platform API as shown in Figure 1.3. This results in all applications getting full access to the sensors irrespective of the parts of the data they need for their functionality. An untrusted application, with such access, can completely violate the user's privacy.

- **Lack of isolation of third-party code in perceptual applications.** Perceptual applications often include third-party code that is not written by the application developer like advertisements provided by ad-brokers as shown in Figure 1.3. As perceptual applications deal with sensitive perceptual data, they must ensure that the untrusted third-party code is correctly isolated and do not get access to the perceptual data unless it needs some parts of the data for its functionality. Without such protection, malicious third-party code may steal and exfiltrate sensitive perceptual data from a benign application. It is usually hard for the perceptual application developers to correctly enforce such isolation. Support for isolated execution of third-party code should be implemented in the perceptual computing platforms themselves so that the application developers can create safe applications by simply invoking the appropriate functions in the framework.

1.5.2 Privacy risks

> 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 because they collect perceptual data from the sensors at a fast rate over a long duration of time. These applications often get access to sensitive perceptual content mixed with benign content. A malicious or buggy perceptual applications leaking such sensitive content may have disastrous privacy consequences. For example, the posture improvement application mentioned in Section 1.2 may leak collected video feeds, read credit card numbers, text on drug labels and computer screens, etc. A simple 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, can surreptitiously gather information about people in the room.

The privacy risks of perceptual applications not only affect the users of those applications but also affect the bystanders i.e. people who are present around the user. For example, perceptual computing hardware such as Google Glass, have already raised significant discussion of bystander privacy —the ability of people around the user to opt out of recording and object recognition. Moreover, certain places like public saunas may want to limit the use of perceptual applications in their premises to protect their patron's privacy.

We categorize the privacy risks posed by untrusted perceptual applications into four different categories based on the required effort and sophistication of the

attacker. We describe each of these categories in detail below.

- **Overcollection of perceptual data.** The simplest privacy risk associated with perceptual applications is overcollection of raw perceptual data. Besides malicious applications, even benign but poorly designed applications may also overcollect sensitive perceptual data. For example, a security-cam application that is designed to detect motion in a room and raise an alarm may collect and store frames from the camera even when there is no motion in the room.

  Identify sensitive items. A more targeted privacy risk than simple overcollection is automated identification and storage of sensitive items. Unlike overcollection of data, such attacks require willing developer participation in violating the user's privacy. For example, a perceptual application may specifically look for items like credit card numbers, license plates numbers, etc. in video frames.

  Aggregation, tracking, and surveillance. Aggregation, tracking, and surveillance are the next class of privacy risks in our hierarchy. Malicious perceptual applications can aggregate large amount of sensitive perceptual data about their users in a short amount of time and use it for tracking and surveillance of the users. In essence, 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 mildly uncomfortable, but it is the accumulation of these across time and space that causes major concerns. Even ignoring

its usage for tracking and surveillance, 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." [16]

- **Semantic Inferences based on perceptual data.** Semantic inferences based on perceptual data is the most sophisticated privacy risk in our hierarchy. Semantic inference does not even require access to raw perceptual data. A perceptual application can perform such inferences with only access to the high-level objects detected in the perceptual data. For example, an application with only access to a rough sketch based on a video frame may be able to infer potentially sensitive gestures, movements, proximity of faces, bodies, etc.

1.6 Towards secure and private perceptual computation

1.6.1 Basic principles

Our main insight is that the perceptual interfaces that platforms provide to the applications are key to providing stronger security and privacy properties as these interfaces control the data getting released by the trusted platform to the untrusted applications. Below we describe the design principles behind the security and privacy protection mechanisms in the perceptual computing platforms that we have built as part of this dissertation.

- **Implement security and privacy protection mechanisms at the platform level.** For wider adoption, the security and privacy protection mechanisms should be implemented as part of the perceptual computing platforms themselves. Individual application developers neither have the expertise nor the time to implement security and privacy protection mechanisms correctly by themselves. However, if the platform supports such mechanisms, application developers can be encouraged and educated to use them.

- **Redesign perceptual interfaces supporting fine-grained privileges.** The permission system in most of the existing perceptual platforms (as shown in Figure 1.3) are too coarse-grained for creating applications that follow the principle of least privilege. Principle of least privilege dictates that each application should only have the privileges that it needs for its functionality [86]. This ensures that the security and privacy risks of an untrusted application will be restricted only to the privileges that the application has. However, existing perceptual platforms tend to have one privilege associated with each sensor. Such coarse-grained privileges do not allow perceptual application developers to minimize the privileges needed for their applications. For example, a perceptual application that requires access to any parts of the visual data stream coming from the camera must have the privileges to access the entire camera feed. The existing perceptual interfaces operate at the raw sensor level and therefore are not amenable for enforcing fine-grained privileges. To minimize the security and privacy risks from untrusted applications, the perceptual computing platforms should redesign their interfaces

that allow application developers to restrict their applications to only access certain parts of the data from the sensors.

- **Automated identification of high-risk applications.** Supporting fine-grained privileges in the perceptual platforms will only be useful if developers use such privileges to restrict their applications' access only to the resources needed for their functionality by following the principle of least privilege. However, prior research [36] has shown that, in existing software markets, the application developers often violate the principle of least privilege and request/use more privileges than the ones that are needed for their application's functionality. This is primarily caused by the fact that the application developers have no incentives to create least-privilege low-risk applications as users do not have the expertise to detect least-privilege violations. This problem can be solved if the market owners provide users access to automated identification mechanisms for high-risk applications violating the least privilege principle and thus encourage them to avoid applications with spurious privileges unrelated to their functionality. This, in turn, will encourage the application developers to follow the principle of least privilege as not doing so may result in lower popularity of their applications.

- **Defense in depth.** As fine-grained privileges alone may not be enough to mitigate all the security and privacy risks of untrusted perceptual applications, the perceptual computing platforms should also use other complementary techniques to minimize such risks. For example, a perceptual platform may

want to minimize the amount of sensitive information available in the perceptual data by applying privacy transforms before allowing the applications to access the data. Alternatively, the perceptual platforms may support applications operating on sensitive perceptual data through invocations of trusted API functions by passing opaque references to the data around without ever revealing the raw data to the applications.

- **Users are the last line of defense.** Despite the usage of multiple protection mechanisms, it might be still hard to mitigate all the security and privacy risks posed by the perceptual applications through technical means alone. Therefore, we must depend on the users to discover and mitigate risks by themselves as the last line of defense. To make the users life easier, the perceptual computing platforms should provide tools to let the users visualize and control the perceptual data an application is getting access to. For example, the perceptual platforms may allow the user to control the data released to an application by tuning parameters like the amount of privacy transforms or by adding/removing the application's access to certain types of sensitive objects (e.g., credit cards, LCD screens, and human faces). The users can use the visualization of the perceptual data going out to the applications provided by the platform to decide when to tune these parameters. The visualization of the perceptual data provided by the perceptual platforms should also be intuitive to the user in order to be useful for such purposes.

1.6.2 Our contributions

As part of this dissertation, we first show that existing perceptual computing platforms like AR browsers are riddled with security and privacy flaws. We present a detailed system-level evaluation of their security and privacy properties, which are often overlooked in the current implementations of perceptual platforms. We start by analyzing the functional requirements that AR browsers must support in order to present AR content. We then investigate the security architecture of the existing AR browsers, focusing on Junaio, Layar, and Wikitude, which are running today on more than 30 million iOS and Android devices. For each functional requirement, we analyze how they implement it using off-the-shelf and custom components; identify design flaws and new categories of security and privacy vulnerabilities unique to AR browsers; and explain how to securely implement the relevant AR functionality.

Next, we explore the following three research questions in order to create a safer perceptual ecosystem. We describe the questions and our answers in detail below.

- How can perceptual platforms redesign their interfaces to the applications for minimizing security and privacy risks?

  We show how perceptual interfaces can be redesigned to support fine-grained privileges to mitigate the security and privacy risks posed by the applications. We also demonstrate that platform support for fine-grained privileges not only minimizes the privacy risks posed by the applications but also frees

the application developers from performing complicated and computation-intensive object recognition on their own. No existing perceptual platform supports support fine-grained privileges. As a result, all perceptual 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 users location, face, or surroundings. Instead of exposing raw sensor data to applications directly, we introduce a new platform 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 perceptual applications for the Xbox 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. This work was originally published in [51].

- Can security and privacy protection be retrofitted under existing perceptual interfaces that can support exiting applications with minimal change?

We find that careful retrofitting of security and privacy protection layers can allow most existing perceptual applications to run unchanged. To this end, we describe the design and implementation of DARKLY, a practical security and privacy protection system for perceptual platforms that can run existing untrusted perceptual applications with minimal changes. 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 security and privacy protection. DARKLY was originally described in [53].

- Will monitoring the usage of perceptual Interfaces by untrusted applications help in identifying anomalous, high-risk applications?

  We demonstrate that monitoring interface usage of applications can lead to detection of high-risk applications violating least-privilege. More specifically, we show how least-privilege violations can be automatically detected by clustering software available from the online markets into peer groups based on its apparent functionality. Such peer groups provide a basis for one

simple, intuitive way of assessing the privileges that are required for providing a certain functionality. Such a basis, once computed by the software market owners, can help users to estimate the risks associated with a piece of software. It can explain how the same privileges may sometimes be natural, yet in other cases be unexpected and suspicious—i.e., why an instant messenger app should get access to the users address book, while an image-editing application might not. We introduce software peer group analysis, a novel technique to identify malicious or unexpected privileges and rank software behavior based on risk. We show that peer group analysis is an effective tool for risk assessment. It provides intuitive, meaningful results, even across different definitions of peer groups and security-relevant behavior. Our evaluation is based on empirically applying our analysis to over a million software items, in two different online software markets, and on a validation of our assumptions in a medium-scale user study.

Chapter 2

Overview of related work

In this section, we provide a brief overview of related works exploring the security and privacy issues in perceptual computing. Detailed comparisons of prior works with the contributions of this dissertation are provided in the corresponding chapters.

2.1 Security and privacy issues in perceptual applications

Some of the security and privacy issues in perceptual applications have been explored in isolation by prior works. However, most of these projects primarily focus on specific ways of exploiting perceptual data under specific threat models but, unlike us, do not perform a comprehensive evaluation of the security and privacy properties of perceptual platforms. For example, Lookout Mobile Security demonstrated how to use a malicious QR code to force Google Glass to connect to an attacker-controlled Wi-Fi access point.[1] Denning et al. [26] showed that lack of proper encryption and authentication in many off-the-shelf consumer robots allows a network attacker to control the robot or extract sensitive data. However, they only

[1] http://www.techweekeurope.co.uk/news/google-glass-security-vulnerability-internet-of-things-122073

considered the network attackers and did not explore the case of untrusted applications. PlaceRaider [97] is a hypothetical mobile malware that can construct a 3-D model of its environment from the phone-camera images assuming that the malware already has access to the camera images.

2.2 Privacy-preserving perceptual computing

There have been several prior attempts at building privacy-preserving perceptual systems. However, all of them are task-specific e.g. only work for the specific task they were designed for (e.g., facial recognition, counting the number of pedestrians etc.) and incur significantly high overhead for being used by the real-time perceptual applications. Unlike these works, our solutions are lightweight and generic.

SciFi [71] leverages techniques for secure multiparty computation for performing facial recognition against a database in a privacy-preserving manner. However, the overhead of SciFi renders it unusable for real-time applications as it can only process one image in 10 seconds. Moreover, SciFi's threat model is quite different as it does not consider the case of untrusted applications.

Prior works often try to detect certain specific sensitive items and remove them from the perceptual data. For example, existing perceptual systems like Google Maps' Street View [95] use simple, ad hoc methods like the blurring of faces and license plates for protecting specific sensitive items. Detecting sensitive objects in surveillance videos by segmenting each frame and transforming them using user-specified policies has been explored be Senior et al. [90]. Dufaux and Ebrahimi [27]

used encryption to protect regions of interests surveillance videos on the network to protect them from unauthorized leakage or modification. However, This requires computationally expensive, offline image segmentation and it is not clear whether perceptual applications would work with the modified videos. A computer vision technique for counting the number of pedestrians in surveillance videos without tracking any single individual was proposed by Chan et al. [17]. Several papers by Sweeney et al. [42, 43, 67] proposed different methods for "de-identifying" datasets of face images.

2.3 Sensor security and privacy

With numerous sensors being packed into computing devices, controlling application access to sensor data has received significant attention from security and privacy researchers. There are primarily four different techniques that can be used to achieve minimize risks of untrusted applications accessing sensitive sensor data: enforce access control on sensors, minimize the amount of sensitive information in sensor data going to the applications, control the usage of sensor data after an application obtains access to it, and let the user visualize sensor data accessed by an application. We discuss related work for each of them below.

Access control. In most modern systems, access control often is implemented through user permissions. Existing operating systems use sensor-specific permissions to restrict access to the sensor data. For example, iOS's permission system prompts the user when an application accesses a sensor for the first time (e.g., a map application first accessing GPS). Android and latest Windows OSes use application-

specific manifests to inform the user about an application's sensor usage during application installation. The application is installed only If the user allows the application permanent access to all the requested permissions . One common problem with the existing permission systems is that they are either disruptive or ask users' permissions without providing the users any context. Prior work by Felt et al [37] has shown that most people simply ignore manifests, and approve the permissions requested by any application. Access control gadgets (ACGs) [81] address these issues by introducing trusted UI elements for sensors, that can be embedded by the applications. Users' interactions with an ACG (e.g., a camera trusted UI) grants the embedding application permission to access the corresponding sensor. The OS ensures the authenticity of the user interaction with each ACG. However, even the ACG-style permission granting is too coarse-grained for perceptual systems because it only allows the users to grant/deny application access to all sensor data for each sensor. However, most perceptual applications only require access to specific parts of the sensor data for their functionality and giving them access to the complete data stream.

Reducing sensitivity of sensor data. Another alternative way of minimize the security and privacy risks of sensor access is to reduce the sensitivity of sensor data (e.g., GPS coordinates) reaching the applications. For example, MockDroid [10] and AppFence [47] inject fake sensor data in order to make the applications execute without access to the real sensor data. Krumm [57] surveys methods of reducing sensitive information conveyed by location readings. However, such simple methods will prevent the applications using the sensor data for their computations

from working correctly. By using well-known statistical methods for calculating the amount of noise to add to a data set, differential privacy [29] can provide strong guarantees against an adversarial application's ability to learn about any specific individual in the data set. While one can imagine using differential privacy for certain types of perceptual computation, in general, most perceptual computations like detecting objects in images are not amenable to differential privacy techniques due to their extremely large input space.

Controlling sensor data usage. Once an application obtains access to sensors through access control mechanisms, the OS can still enforce information flow control based approaches to control/monitor an application's usage of the sensitive data as shown in TaintDroid [31] and AppFence [47].

Visualizing sensor data access by each application. *Sensor-access widgets* by Howell and Schechter [48] introduced the idea of showing the outputs of sensors to the users in order to make them aware of the perceptual data released to the applications. However, their widgets always display the entire camera feed to the users because applications in existing perceptual system allow the applications to access the visual inputs without any restrictions.

Higher-level abstractions for sensors. CondOS [18] introduced the notion of Contextual Data Units (CDUs) as a new high-level OS abstraction for accessing the sensor data along with its context. However, they did not choose a set of concrete CDUs that can support a wide variety of real-world applications. Also, they did not address the privacy risks arising from untrusted applications having access to CDU values. Koi [45] provides abstractions for matching location data without requiring

applications access raw GPS coordinates. However, Koi's approach is limited to only GPS coordinates and difficult to generalize across all sensors.

2.4 Protecting bystander's privacy from perceptual applications

As perceptual applications like facial recognition is becoming commonplace both for law enforcement purposes [13] and personal usages [4], the privacy of bystanders who themselves are not using perceptual applications but accidentally came into the field of view of some perceptual application running on others' devices. CV Dazzle [23] suggests innovative make up patterns that one can apply to her face to avoid being detected by facial recognition software. However, such make up, besides being cumbersome to apply, may attract significant human attention.

ObscuraCam [68] is a smart camera application that allows the application owner to remove/blur the auto-detected faces in a image selectively before storing or sharing it. Szegedy et al. [96] showed that every image, for which a object detector using deep neural networks can detect objects correctly, can be tweaked slightly in a visually imperceptible way to create a new image where the object detector will not work. Such techniques can also be used by a user to perturb the image before sharing it to prevent facial recognition from working on the image. However, as the perturbation is specific to the facial recognition algorithm, recognition may still be possible with other algorithms. PlaceAvoider, by Templeman et al., allows the perceptual device owners to automatically prevent the sensors (e.g., camera) from working in certain sensitive spaces (e.g., bathrooms, bedrooms, etc.). Unfortunately, all these techniques require cooperation of the perceptual application

owner which may not be always feasible.

Chapter 3

Security and privacy flaws in existing perceptual platforms

3.1 Introduction

Most existing perceptual platforms like Microsoft Kinect SDK and OpenCV do not protect the raw perceptual data from untrusted applications and therefore do not provide any protection against them. However, Augmented Reality (AR) browsers are one of the few perceptual platforms that provide a certain level of isolation and protection from untrusted AR content. In this chapter, we evaluate the security and privacy properties of AR browsers and show that they suffer from serious flaws. We also provide guidelines for fixing these flaws in a principled manner.

Augmented reality (AR) technologies enhance users' perception of the world by blending interactive virtual objects with the visual representation of actual objects in real time [5, 6]. Traditional AR applications range from medical visualization to aircraft navigation, but only recently have consumer mobile devices become sufficiently powerful to run AR software.

AR applications have three stages: sensing input, transforming sensed objects (e.g., adding virtual objects), and rendering the transformed objects to the

user. Modern AR platforms ease the burden of implementing these tasks. By far the most popular platforms are *AR browsers* like Junaio, Layar, and Wikitude, available as SDKs or standalone mobile apps. Junaio has more than 20 million users and over 20,000 content developers who have created more than 210,000 AR "channels" [54]. Layar has 1.5 million users and 9,000 content developers [59]. Wikitude has 13 million users [101] and over 30,000 content developers.

All existing AR browsers are based on Web browsers and are similar to them in the sense that they, too, fetch and display interactive content from websites ("channels," in AR parlance). In addition to rendering HTML and executing JavaScript, AR browsers provide support for the three key tasks necessary for AR functionality: sensing, transforming, and displaying transformed objects. They enable AR channels to (1) access sensors on the mobile device, including the onboard camera and GPS location, (2) create and manipulate a variety of 2D and 3D interactive virtual objects, and (3) display virtual objects on top of the camera feed, realistically blending them with real objects. The resulting AR content combines image recognition, geolocation, interactive virtual objects, conventional Web content, and control code written in JavaScript (see an example in Figure 3.1).

The basic architecture of AR services is shown in Figure 3.2. From the security and privacy perspective, its key aspect is that the AR browsers provide augmentation mechanisms, but the actual AR content comes from channels created by independent developers. Just like a conventional Web browser is an interface between the user and Web content from independent websites, an AR browser is an interface between the user and independent AR content.

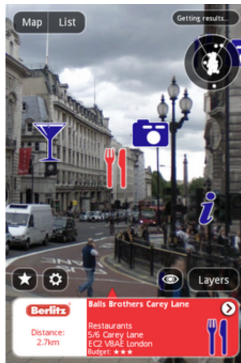


Figure 3.1: A Layar-based mobile app [33].

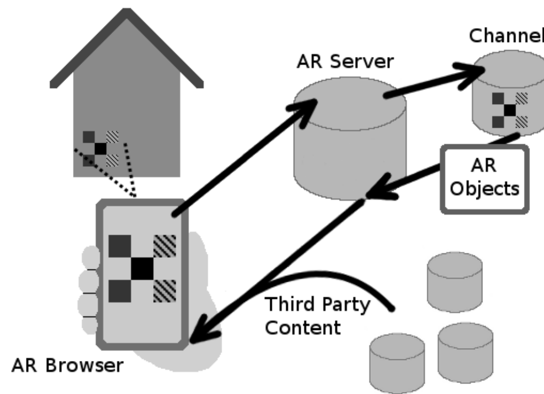


Figure 3.2: Basic structure of AR services.

A major difference between Web browsers and AR browsers is the business model. Web browsers are typically part of the standard software distribution, and their developers are paid by the licensing fees from OEMs and OS owners and by the search engines. This model works because there is already a wealth of Web content. AR browsers, however, need a different model because there is not much AR content available today. Their sources of revenue include advertising injected into AR content, registration fees from content developers, and revenue sharing for

paid content. This business model has an impact on the architecture of AR services: unlike Web content, which is accessed directly from the Web browser, requests to load third-party AR content must go through the AR service provider, as shown in Figure 3.2.

Our contributions. We perform the first systematic analysis of the security and privacy properties of AR browsers and how they differ from Web browsers. Untrusted AR content presents new, unique types of threats, yet—in contrast to the Web-browser specifications—the latest Augmented Reality Markup Language (ARML) specification [69] barely mentions security or privacy, and they are often overlooked in the design of the existing AR browsers.

We start by analyzing the functional requirements needed to support sensing, transforming, and rendering of AR content. These include new ways of combining AR objects and conventional HTML content from multiple origins, new APIs for accessing objects outside the browser, new mechanisms for controlling the display of AR and HTML objects, and new ways of launching content (e.g., by scanning a picture or QR code).

Then, for each functional requirement, we investigate how it is implemented by the existing AR browsers, all of which are based on embedded Web browsers such as WebView. Web browsers do not provide system support for AR functionality, forcing AR browsers to resort to ad-hoc cross-origin mechanisms, APIs that open holes in the browser sandbox, custom techniques for composing visual content from different origins, and non-standard delegation schemes for authentication

credentials.

Architectural flaws in these mechanisms result in security and privacy vulnerabilities. We explore the threat model of AR browsers and demonstrate several entirely new categories of threats caused by the AR browsers' unique combination of high-volume visual data gathering, image-triggered code execution, outsourced image processing, and merging images from the onboard camera with third-party content. For example, (1) individual-specific items such as license plates can automatically launch malicious AR content, enabling fully automated stalking and tracking; (2) malicious AR channels can abuse image-triggered code execution; and (3) a conventional webpage can hijack the AR browser installed on the user's mobile device and use it to gain unauthorized access to the device's camera and GPS without the user's permission. We also show how the architecture of AR browsers results in the amplification of existing threats. For instance, (4) a malicious AR channel or even a malicious online ad can exploit the AR browsers' mechanisms for combining non-HTML AR objects with HTML content to perform universal cross-site scripting attacks against any origin; (5) outsourced image recognition accidentally leaks private data (e.g., images of credit cards, computer monitors, etc.) to the AR server and any network eavesdropper; (6) new mechanisms for visually composing different types of AR content can be exploited for clickjacking; (7) a malicious channel can exploit incorrect delegation of authentication credentials to steal cookies belonging to any origin.

For each design flaw, we present our recommendations. Unfortunately, few of the problems we identified can be solved simply by adding AR support to

HTML5. Extending the same origin policy to AR tags is non-trivial, outsourced image processing and image-triggered code execution will continue to present security and privacy challenges, etc. For each functional requirement of the AR browsers, we explain which features and system abstractions are needed to implement it properly. These recommendations include concrete fixes for specific vulnerabilities and guidelines for securely re-engineering the architecture of AR browsers.

In summary, we present the first security and privacy specification for AR browsers, a popular augmented-reality technology running on millions of devices.

3.2 Related work

Azuma et al. [5,6] identified three major properties of AR systems, exhibited by all AR browsers in our study: combining real and virtual objects, real-time interactivity, and support for 3D blending of virtual and real objects. Several papers suggested adding augmented reality to mobile applications such as tour guides [1, 34]. Spohrer et al. [94] explored the idea of associating information with real-world objects using "WorldBoard channels." Just like AR channels analyzed in this paper, "WorldBoard channels" can display HTML-encoded information overlaid on real-world objects. Kooper et al. [56] used the term "real world-wide web" for the combined information space created by merging real-world objects with the HTML content of the WWW. ARML [69] is a proposed standard for defining geospatial AR objects through XML.

Roesner et al. surveyed various security concerns arising from the widespread use of AR technologies [80]. By contrast, we analyze the technical architecture of

popular, deployed AR platforms. With the exception of clickjacking and general privacy concerns, none of the issues we discovered are mentioned in [80].

Several recent papers focused on privacy concerns arising from the unrestricted access to sensor data by untrusted third-party AR applications. Darkly, by Jana et al., uses access control, algorithmic transformation of image features, and user audit to prevent certain privacy violations by applications based on the OpenCV computer vision library [53]. D'Antoni et al. [24] and Jana et al. [51] show how to restrict AR applications' access to sensor data by adding fine-grained permissions to the OS. These systems are concerned with protecting users from untrusted applications, whereas we investigate whether and how trusted AR applications protect users from untrusted content (i.e., our threat model is similar to the standard threat model of Web browsers).

Some of our attacks involve pictures or QR codes placed in a public area to trick AR browsers into launching a malicious AR channel. Lookout Mobile Security used a QR code to force Google Glass to connect to an attacker-controlled Wi-Fi access point.[1] Both attacks employ malicious QR codes, but the similarities end there. The attacks described in Section 3.7.2 exploit the deficiencies of user interfaces in AR browsers, not software vulnerabilities.

Clickjacking attacks against conventional Web content were analyzed in [49, 83,84]. In Section 3.9.2, we explained that our clickjacking attacks and defenses are somewhat different because of the architectural differences between Web browsers

[1] `http://www.techweekeurope.co.uk/news/google-glass-security-vulnerability-internet-of-things-122073`

and AR browsers.

3.3 AR services

AR services are deployed by AR *service providers* such as Junaio and Layar. These companies supply AR client software (we use the term *AR browser*) to users and maintain dedicated AR servers through which users access third-party AR content (see Figure 3.2). AR *content providers* are independent developers who create AR content, host it on their own servers, and register this content with AR service providers. We use the term *channel* generically for any AR content, but the actual terminology differs from service to service (e.g., channels are called *layers* in Layar).

By analogy with conventional Web, AR service providers are similar to Web-browser developers, while AR channels are similar to Web applications. There are important differences, however. AR providers make money by charging for SDK licenses, features such as cloud storage for AR channels, and per-user fees from third-party apps that connect to their services. All providers analyzed in this chapter allow a limited use of free channels, but some charge for commercial channels and/or may insert banner ads into free channels. Therefore, they typically require that browsers initiate access to third-party channels via providers' own servers.

3.3.1 Functional requirements

Access to native resources on the user's device. The AR browser must have access to the onboard camera and GPS location to (1) recognize images and locations that launch AR content, and (2) to correctly add AR objects to the user's visual environment.

Support for interactive, non-HTML AR content. In addition to HTML content such as images and text, AR content may include 2D and 3D models and animations that cannot be described in HTML. AR channels thus include service-specific XML or JSON defining how to place and render these objects.

Image-triggered code execution. AR browsers access content in non-standard ways: they send images from the device's camera to their servers, which attempt to recognize certain pictures and QR codes and automatically launch the associated AR channels.

Outsourced image processing. Image recognition is a computationally heavy task that may not be feasible on low-powered mobile devices and often involves proprietary algorithms. Furthermore, image-based code execution requires the server to extract the trigger image from the camera feed and match it against a proprietary database of registered images. Therefore, AR browsers send images from the phone's camera to the AR provider for processing.

Visual composition of AR content. The AR browser is responsible for constructing a visual stack that combines non-HTML AR content, such as interactive 3D models,

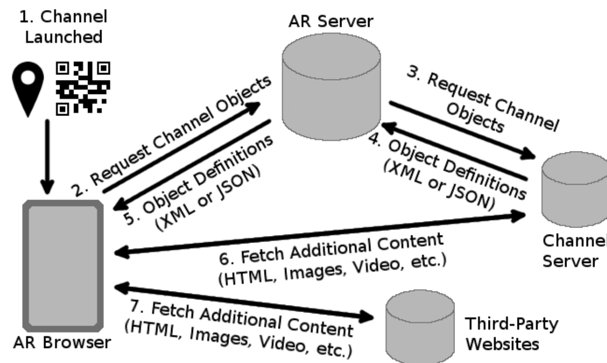


Figure 3.3: Architecture of a typical AR service.

with HTML content from multiple origins (e.g., online ads) on top of the camera feed.

Indirect retrieval of AR content. Instead of directly fetching AR content from its developers, AR browsers typically submit requests via the AR provider's server. This enables providers to charge registration and usage fees, inject advertising, etc.

3.3.2 Components of AR services

The components of a typical AR service and their interactions are shown in Figure 3.3.

AR browsers. Figure 3.4 shows the generic architecture of an AR browser. All AR browsers include (1) one or more instances of an embedded Web browser such as WebView, (2) a "native" component that has direct access to resources such as the camera and GPS location via the mobile OS, and (3) ad-hoc mechanisms for gluing these components together.

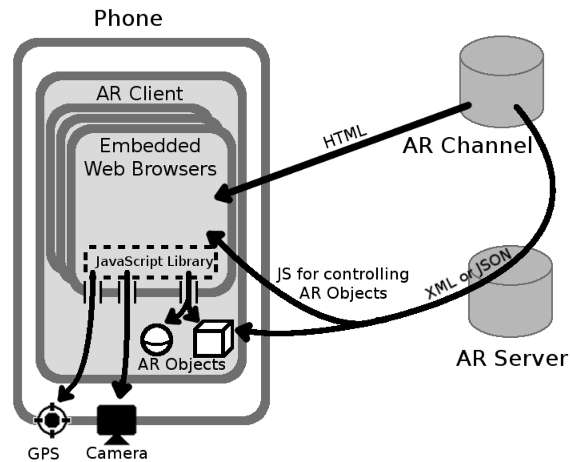


Figure 3.4: Architecture of an AR browser.

AR channels. An AR channel is roughly similar to a website. It defines an augmented reality experience by specifying AR content to display and how to display it. This content may include AR objects linked to a geolocation ("points of interest" or POI), HTML pages, audio, video, etc., as well as JavaScript to control these objects. The channel may also specify the actions to take when a certain object comes into view or is clicked by the user.

For example, an AR channel may overlay historical pictures when viewing certain landmarks,[2] show reviews for nearby restaurants,[3] or control an avatar running around the scene.[4] A channel may directly incorporate third-party content—for instance, include online ads in its HTML—or instruct the AR browser to load a third-party webpage when the user performs a certain action.

[2] `http://layar.it/YuDzik`
[3] Wikitude Restaurants
[4] `junaio://channels/?id=127275`

A user launches a channel by selecting it from a list provided by the AR browser (based on the geolocation or most popular channels), or by scanning an image.

AR servers. As explained in Section 3.3.1, requests to load a channel are sent by the AR browser to the AR provider's server, not directly to the channel server (see Figure 3.3). Each request includes some combination of the channel's id, the location of the device, and other data. The AR server forwards the request to the relevant channel's server, which is a server controlled by the channel owner and registered with the AR provider. The AR server may also handle the authentication of users to channels (Section 3.10). The response from the channel with the XML or JSON definitions of AR objects is forwarded via the AR server, too. Subsequent requests may be sent by the browser directly to the channel server.

3.3.3 Specific AR browsers

We focus on the most popular AR browsers. **Junaio** is an AR browser developed by Metaio. It is designed to augment both print media and geolocation-based environments (Figure 3.5). **Layar** focuses primarily on adding AR features to print media such as magazines and newspapers (Figure 3.6), but also supports geolocation-based AR. AR content for Layar is served by *layers*, but we will refer to them as *channels* for terminological consistency. **Wikitude** is another AR browser, but some of its features did not execute correctly in our testing, thus we discuss only the features we were able to evaluate.

Unlike HTML, AR content is browser-specific, i.e., a Junaio browser can

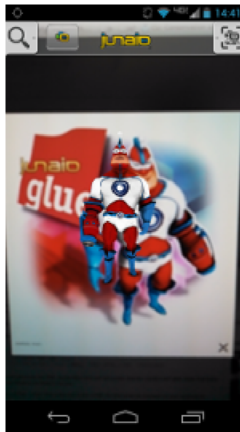


Figure 3.5: Examples of a Junaio channel showing a 3D model placed over the Junaio logo.

only display Junaio channels, etc. Wikitude and the Open Geospatial Consortium have proposed the Augmented Reality Markup Language (ARML) standard to unify the format of AR objects with XML [69].

3.4 Threat model

We are concerned with five classes of attackers.

AR attackers. An AR attacker is the equivalent of a standard Web attacker. He controls the malicious content of his AR channel and may trick or entice users into visiting it. He cannot observe users' network communications with other destinations, nor execute any code on their machines other than JavaScript served by his own channel. For image-triggered channels, the attacker can associate any image with his channel. Some—but not all—of the attacks described in this chapter require the attacker to place physical trigger images in a public place (e.g., as a sticker

Figure 3.6: A Layar channel running on top of a scanned magazine page. The AR objects are circled. Clicking any of the colors below the 3D watch model changes its color. The user can also add the watch to his or her shopping cart.

on a wall).

Curious AR services. We assume that AR browsers are benign and secure (the issues raised by malicious mobile apps are vast and well beyond our scope), but we do investigate privacy risks caused by user-specific visual data collected by AR services.

Ad attackers. AR channels can include third-party content such as syndicated ads. An "ad attacker" tricks a trusted website or AR channel into incorporating his malicious content, e.g., via ad brokers. We assume that ads can run arbitrary JavaScript, but are confined into iframes when rendered by the AR browser.

Web attackers. While the focus of this chapter is on malicious AR content, we also investigate how the mere presence of AR browsers on the device can be exploited by conventional Web attackers (Section 3.5.2). A Web attacker controls his own website and attempts to lure users to it via enticing content, ads, phishing, etc. Like the AR attacker, the Web attacker cannot observe network traffic between the user

and other sites.

Network attackers. We briefly analyze privacy threats posed by network attackers. Either through man-in-the-middle attacks or by being on the same network as the victim, a network attacker can listen in on the communications between the AR browser and the AR provider, AR channel owners, and third-party servers.

3.5 Access to native resources

AR browsers cannot function without access to the camera and GPS location since both are required to launch channels. Furthermore, individual AR channels need access to these native resources in order to correctly add AR objects to the user's camera view. Consequently, all AR browsers equip JavaScript with some form of access to native device resources outside the Web-browser sandbox. Script access to AR objects is also required by the ARML 2.0 specification [69, Section 9.1] and therefore must be supported by all compliant AR browsers.

Another common functionality is launching AR browsers via custom URLs. This is needed for interoperability [76, Section 5]: for example, one AR browser may launch another AR browser to render proprietary content that is not supported by the first browser.

The control code of Junaio channels is written in JavaScript. Junaio executes it in an embedded WebView browser extended with custom interfaces that allow JavaScript to grab camera images, read and change the Junaio-reported geolocation, control the device's light, make requests to the channel server, open conventional

Web-browser windows, or load a different channel.

Technically, these interfaces are accessed via AREL, a JavaScript library from Junaio that encodes commands in pseudo-URIs. For example, *arel://media/website/?action=open&external=false&url=http%3a%2f%2fwww.google.com* instructs the Junaio app to launch Google.com in a conventional browser. There are several techniques to pass this pseudo-URI from WebView to the Junaio app. The standard technique is to push the URI to the global "commandQueue" and set *window.location = "arel://requestsPending"*. The Junaio app intercepts the URL load event, reads the command off the queue, and performs the requested action.

Layar supports a more restricted set of native-access capabilities. For example, a channel can be invoked via a *layar://[channelname]* pseudo-URI.

3.5.1 Doing it wrong

The custom APIs described above are generic architectural features, universally supported by AR browsers and independent of any specific AR content. They effectively open holes in the Web-browser sandbox, intended to support native access by the channel's own JavaScript. Unfortunately, these externally added APIs are not protected by the same-origin policy, in part due to the lack of system support (WebView does not provide any way to restrict the origin of custom interfaces). Consequently, they can be accessed by any HTML content regardless of its origin. For instance, in Junaio on Android, any iframe can bypass AREL and execute native commands directly, without user permission, by setting *window.location* to the

corresponding pseudo-URI.

3.5.2 Risks

In this section, we are concerned with (1) conventional Web attackers, whose malicious pages are viewed by mobile users in conventional browsers, (2) "ad attackers," whose untrusted HTML is incorporated into trusted AR channels but confined into iframes, and (3) AR attackers, who directly control malicious AR channels.

Hijacking AR browsers to bypass OS access control. AR browsers must have access to the camera, but conventional webpages typically don't, unless explicitly authorized by the user. Unfortunately, the AR browser's access rights can be hijacked by malicious webpages to gain access without the user's permission.

Suppose the user has the Junaio app installed on his Android phone. The user accidentally visits a malicious webpage in the regular Web browser (e.g., Android's default system browser) by clicking on an ad, a link in a spam message, etc. The malicious page contains a URL of the form *junaio://channel*=... and a script in the page forces the browser to open this URL. This generates an Android intent, which automatically starts the Junaio app and launches any channel chosen by the attacker, e.g., the attacker's own channel. Like all Junaio channels, the attacker's channel automatically has access to the device's camera, can take pictures of the user and its surroundings, etc. Layar, too, can be automatically launched from a conventional webpage via a *layar://[channelname]* pseudo-URI.

This attack completely bypasses OS access control. Even though the user granted camera access only to Junaio or Layar, this access has now been hijacked by a conventional webpage. The attack can even be stealthy. Having grabbed images from the camera, the attacker's channel can relaunch the regular Web browser and immediately redirect the user to the page he was initially browsing.

The ability to automatically launch an AR browser is required for interoperability between AR browsers [76]. Therefore, the presence of a single AR browser that can be launched in this fashion on the user's device is exploitable by conventional Web attackers.

Hijacking native-access rights by malicious ads. Because native-access rights are not restricted to the channel's own origin, any iframe can hijack them. In Section 3.6.2, we describe how native-access capabilities can be used by a malicious ad to perform a cross-site scripting attack against any origin of its choosing.

Furthermore, malicious third-party iframes included into a trusted channel can take redirect the entire AR browser to malicious content. For example, in Layar, a script in an iframe can use a *layar://* command to switch the browser to a different channel. In Junaio, the switchChannel command in AREL (also accessible from an iframe) has the same effect. This can be exploited for undetectable phishing: an iframe included in a trusted channel's HTML can automatically switch the browser to a visually indistinguishable malicious channel.

Abusing native-access rights by AR channels. The ability of AR channels to access resources outside the browser sandbox presents privacy risks to their users.

In Junaio, as long as the channel's transparent overlay (Section 3.6) continues to run in the background, it can surreptitiously grab images from the camera and send them to the channel server even after the user moved away from the place where he launched the channel. The user's location can be tracked in a similar fashion in Junaio, Layar, and Wikitude.

3.5.3 How to do it right

To prevent conventional webpages from gaining unauthorized access to the camera and other resources by launching the AR browser and directing it to the attacker's channel, the user should be asked for confirmation whenever the AR browser is invoked automatically. This presents serious interface-design and usability challenges and calls for further research.

Interfaces to native resources must be protected by the same-origin policy, lest they are hijacked by untrusted iframes. Recent solutions to the problem of unauthorized native access by third-party origins in mobile apps, e.g., NoFrak by Georgiev et al. [39], may be applicable to AR browsers. Alternatively, embedded Web browsers such as WebView can be re-engineered so that the same-origin policy applies to native-access interfaces.

Additionally, AR browsers could be re-designed to support fine-grained native-access permissions. Instead of wholesale access to a particular resource, the channel would be restricted to a limited set of actions outside the sandbox. For example, it could only access the camera feed via pre-defined system abstractions such as "recognizers" [51] for specific objects.

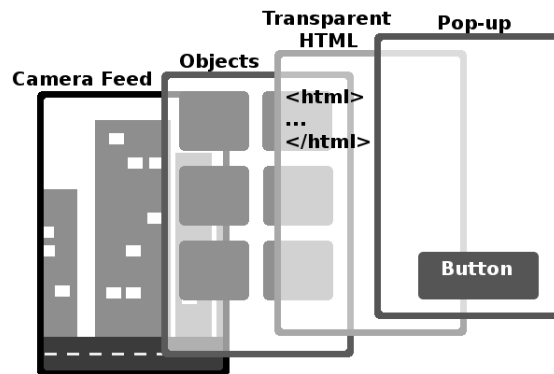


Figure 3.7: Junaio's visual stack. AR objects are on top of the camera feed, the transparent overlay on top of the objects. If an object is clicked, a pop-up appears at the very top.

3.6 Support for non-HTML AR content

In addition to HTML content such as images and text, interactive AR content includes videos, animations, and 2D and 3D models with unique visual presentation requirements. Because these AR objects cannot be described in HTML alone, existing AR services rely on XML or JSON definitions to specify how to place and render these objects, and on JavaScript to control these objects from inside the embedded Web browser.

Junaio. In Junaio, AR objects are defined in the XML page returned by the channel server. Junaio supports floating clickable objects ("points of interest"), 3D models, floating pictures, movies, and 360-degree panoramas. The Junaio browser renders these objects in the visual stack shown in Figure 3.7.

On top of the AR objects, Junaio places a transparent window implemented using WebView (Android) or UIWebView (iOS). We call it the *transparent HTML*

overlay. This overlay provides GUI functionality to channels and enables them to control AR objects outside WebView via special browser interfaces and a custom JavaScript library called AREL (Section 3.5). These interfaces can be used to create, destroy, animate, move, or resize AR objects, to read and modify their parameters such as id, name, geolocation, and the associated pop-up, and to handle events based on channel state, object state, or user's interaction with the object (e.g., channel ready, object loaded, sound finished playing, object rotated, etc.).

The URL of the transparent overlay is specified in the XML page and may belong to a different origin than the AR channel itself. This URL cannot be viewed by the user. The channel—and any third-party content included in the channel—can also supply JavaScript that will be executed inside the transparent page.

Clicking a link in the transparent overlay loads the destination in the same window, replacing the old page. JavaScript in the transparent overlay can also open an opaque window with a conventional embedded Web browser. Another way to open an opaque window is via a pop-up (Section 3.6.1). JavaScript continues to run in the background after opening the window.

Layar. The Layar browser displays AR objects on top of the visual feed from the device's camera (Figure 3.8). The objects can be HTML webpages, 2D images, 3D models, or videos, and can have actions associated with them, such as placing a phone call, sending an SMS or email, launching a website, loading or refreshing channels, sharing the channel on Facebook or Twitter, and loading movies and music. Actions are specified in the object definition via pseudo-URIs such as 'tel:',

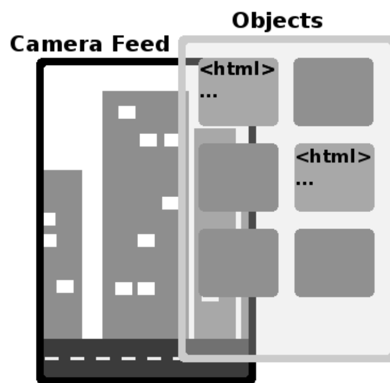


Figure 3.8: Layar's visual stack. AR objects, which can include HTML pages, are overlaid on the camera feed.

'sms:', 'mailto:', 'layar://' , 'layarshare://', etc.

Wikitude. Wikitude is architecturally similar to Junaio. AR content includes a transparent webpage that shows a GUI and controls AR objects via a custom JavaScript library called ARchitect. Object types include *HtmlDrawable*, intended to display HTML content. HtmlDrawables have an evalJavaScript function that can be used to execute JavaScript inside a drawable (it worked only sporadically in our testing on Android 4.1.2).

3.6.1 Doing it wrong

Web browsers take great care to confine untrusted Web content. The same-origin policy (SOP) ensures that content from a given origin—defined by the protocol (HTTP or HTTPS), domain name, and port number—cannot access the non-trivial attributes of any content from a different origin [93]. Web browsers provide abstractions such as iframes for confining content from untrusted origins, as

well as structured cross-origin mechanisms (e.g., postMessage), which, if used correctly [92], enable content from different origins to communicate.

AR browsers face a similar challenge because AR channels may combine content from different origins. Interactive, non-HTML AR objects make this challenge much harder because they are must be described in XML or JSON, which are not governed by the SOP. Therefore, any Web content included in XML definitions must be sanitized to prevent cross-origin attacks. None of the existing AR browsers apply such sanitization.

For example, the XML definition of an AR object in Junaio can have a *pop-up* field with a textual description and an array of buttons. Each button contains either a URL, or JavaScript code. When such an object is clicked, a partially transparent window with the pop-up's description and buttons is opened on top of the transparent overlay (Figure 3.7). When a button is clicked in the pop-up, the associated URL is loaded in an opaque window. If the pop-up contains JavaScript instead, this JavaScript is executed in the transparent overlay—even if the origin of the content in the overlay is different from the origin of the channel that provided the script.

Furthermore, AR browsers provide ad-hoc interfaces for communication between layers of the visual stack (Section 3.5). The interfaces used by HTML content to manage AR objects outside the embedded Web browser are not protected by the SOP.

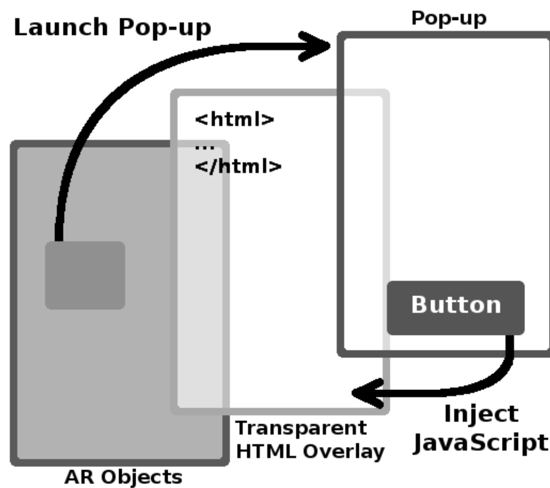
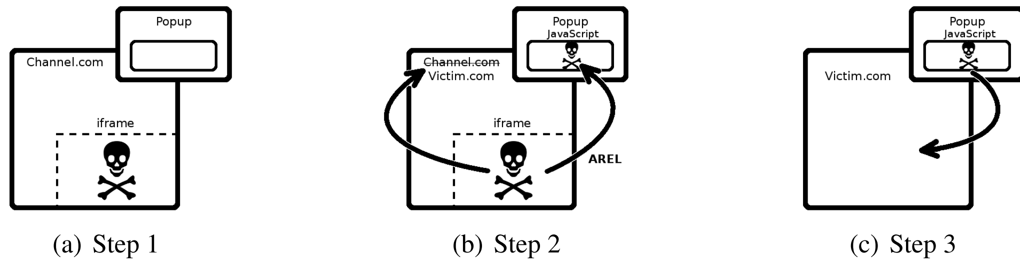


Figure 3.9: Cross-site scripting (XSS) in Junaio

3.6.2 Risks

In this section, we are concerned with AR attackers, who may incorporate trusted content into their malicious AR channels, and "ad attackers," whose malicious content (e.g., online ads) is incorporated into trusted AR channels, but confined into iframes.

Cross-site scripting. As described above, an AR object in Junaio may be associated with a pop-up button, which can inject JavaScript into the transparent webpage overlaid on the channel. This setup opens a hole in the same-origin policy. A malicious channel can specify any origin for the transparent page and associate an arbitrary script with the button. When the button is clicked, this script is injected into the transparent page and gains unrestricted access to all content from this page's origin—see Figure 3.9. This cross-site scripting (XSS) vulnerability can



(a) Step 1 (b) Step 2 (c) Step 3

Figure 3.10: Universal XSS vulnerability in Junaio.

be exploited, for example, to modify the DOM of the victim page or steal cookies.

HtmlDrawable objects in Wikitude contain an even simpler XSS vulnerability. A malicious channel can (1) specify any URL for an HtmlDrawable object, and (2) use evalJavaScript to inject an arbitrary script into this object. In effect, HtmlDrawable objects present an easy-to-use, structured API for performing XSS attacks.

Universal cross-site scripting in Junaio. The above XSS attacks assume that the channel is malicious. Unfortunately, even if (i) the channel itself is benign, (ii) all untrusted, third-party content such as online ads is correctly confined to iframes, and (iii) the embedded Web browser running the channel's HTML correctly enforces the SOP, confined third-party content in Junaio can perform XSS attacks against *any* origin of its choosing.

Consider a benign Junaio channel that includes an AR object with a popup button and suppose that the channel's transparent HTML page contains an ad in an iframe (Figure 3.10(a)). Malicious JavaScript hidden in such an ad can (1) use AREL commands (Section 3.5) to change the script associated with the pop-

up button to the attack script, and (2) change the URL of the transparent overlay to the victim page (Figure 3.10(b)). When the button is clicked, the attack script will be executed in the victim page (Figure 3.10(c)). These architectural features thus amount to a *universal XSS vulnerability*: a malicious ad can pick any origin whatsoever and inject an arbitrary script into it.

3.6.3 How to do it right

Quick patches. The cross-site scripting vulnerabilities described above are caused in part by the fact that the origin of HTML incorporated into an AR channel may be different from the channel's own origin. A plausible defense is to require the AR browser to ensure that the two origins are the same, but this requires the AR browser to reason about the origins of content specified in custom XML definitions (as opposed to HTML). AR browsers can no longer rely on WebView to manage the same-origin policy and must correctly replicate a complex piece of Web-browser functionality.

Another defense is to sanitize XML so that it does not contain scripts, which is a notoriously difficult problem [8] and, in the case of AR browsers, requires careful reasoning about origins that normally would have been the responsibility of the Web browser. Both defenses disable important features of AR browsers and may break existing applications.

In Wikitude, where evalJavaScript allows channels to inject scripts into an HtmlDrawable regardless of its origin, restricting the origin is not feasible because HtmlDrawable is intended to display content from origins other than the channel

itself. The evalJavaScript interface should simply be removed.

Principled solutions. The root cause of many security holes described in this section (but not those in the other sections) is that AR objects cannot be described in HTML, thus AR browsers must use custom mechanisms to enable HTML content to control these objects. Standardizing AR object description languages, including them in HTML5 via either tags, or a special document type, (e.g., *channel*), and adding support for these new HTML5 features into embedded Web browsers like WebView would allow AR content to execute entirely within the Web browser, eliminating the need for XML and some of the ad-hoc browser interfaces.

Unfortunately, assigning origins to these tags is not trivial. In the existing AR browsers, all AR objects are treated as if their origin is the domain where the main AR channel is hosted. Since any of these objects may contain JavaScript, this is extremely dangerous.

The alternative, which is to extend the same-origin policy to AR tags, is problematic as well. These tags are intended to support 3D models, animations, UI elements, etc. which may come from different domains but are intended to work smoothly together to produce a unified AR experience. A naive extension of the SOP would isolate the AR HTML tags based on their domains, but this would prevent them from communicating. The developers would then have to implement cross-origin communication mechanisms, which is fraught with peril [92]. Enforcement of the SOP is complicated even further by the fact that several of these new tags may need plugins to be rendered (similar to Flash).

Lacking HTML5 support for AR, embedded Web browsers such as WebView can at least provide SOP-protected APIs that (1) render arbitrary objects on top of camera images, and (2) let JavaScript inside WebView control these objects. This would support safe implementation of an important subset of AR functionality.

3.7 Image-triggered code execution

The ability to scan their surroundings and to recognize and track images is fundamental in AR browsers [69, Section 7.5.1.2]. This enables new methods for invoking AR content. In Junaio and Layar, AR channels can be associated with pictures or QR codes and launched simply by scanning the corresponding image.

3.7.1 Doing it wrong

When the user is scanning his surroundings through the Junaio or Layar browser, the AR service is continuously analyzing the camera feed. As soon as it recognizes an image associated with some channel, it automatically launches and executes the channel's content, without any confirmation prompts. The user is not given an opportunity to preview the URL or any other information about the content, with one exception: for QR codes (but not pictures), Layar previews the URL by showing it as a button before launching the channel. Unfortunately, its URL parser is broken (Figure 3.11). For example, if the URL in the QR code is *http:////attacker.com*, it will not be displayed in the preview, but the browser will launch AR content hosted at *http://attacker.com*.

In Junaio, *after* a channel is fully loaded, the user can see its description and

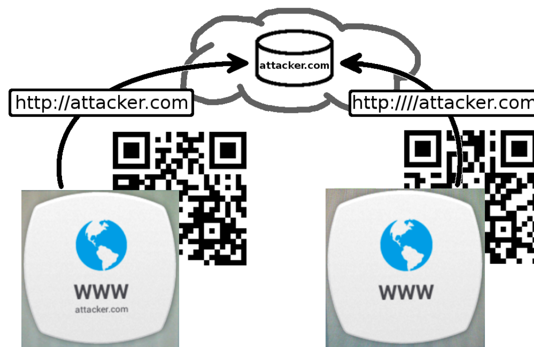


Figure 3.11: Both codes launch the same channel, but Layar fails to parse the code on the right and does not show the URL.

the developer's name.

3.7.2 Risks

In this section, we are primarily concerned with AR attackers who can choose any picture or QR code as the automatic trigger for their malicious channels. For some (but not all) attacks, the attacker also needs to physically place these images in public places, possibly in proximity to other channels' trigger images.

Stealthy tracking. Because AR services do not vet pictures associated with AR channels, they can be used for automated stalking and tracking. Layar's image recognition algorithm is sufficiently precise to distinguish between different license plate numbers. An AR attacker can register a Layar channel associated with the photo of a specific license plate. Whenever any Layar user scans his surroundings and the license plate is prominent in the camera's view, the channel would get launched automatically and the plate's location, along with its entire visual en-

vironment, will be sent to the channel's owner, enabling him to track the plate's movements. Other sensitive items can be tracked in a similar fashion.

Crucially, this attack does not require the attacker to distribute physical images associated with his channel.

Exploiting automatic channel launching. As soon as the image is recognized by the AR service's (black-box) recognition algorithm, the code of the associated channel executes automatically, without user confirmation or channel identification. If the scanned image contains sub-images or a familiar image in an unusual environment, the user cannot know ahead of time which channel will be launched. If the images are similar, yet subtly different from each other, the user may not be able to tell which channel they launch.

Image recognition algorithms suffer from false positives and are inevitably nondeterministic from the user's point of view [102]. Unfortunately, user interfaces of the existing AR browsers are derived from the embedded Web browsers on which they are based and do not inform the user about spurious matches and other potential problems with visual identification.

This can be exploited by an AR attacker in two ways: (1) register an image trigger that is very similar to an image already associated with a trusted channel, or (2) combine the image associated with a malicious channel with a trusted channel's image into a single composite image. In either of these scenarios, the AR browser may be tricked into automatically launching the malicious channel when scanning a physical image placed by the attacker (e.g., on a building wall, bus shelter, etc.).

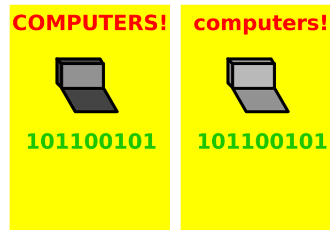


Figure 3.12: Depending on the angle, each poster nondeterministically launches either its own channel, or the channel associated with the other poster.

In Layar, the same picture may be associated with multiple channels. For example, we have been able to register our channel with the same movie-poster image as one of Layar's demo channels. If there are multiple channels associated with a picture, the user may open a menu in the corner to see channel names and switch between them. It is possible, however, to create visually similar images that automatically and non-deterministically launch different channels without the browser presenting the channel selection menu to the user. Each poster in Figure 3.12 nondeterministically launches the channel associated with the poster or the (completely different) channel associated with the other poster. At many viewing angles, the channel selection menu is not offered.

Furthermore, a malicious channel can suppress the channel selection menu using the native-access capabilities described in Section 3.5.2. A *layar://[channelname]* pseudo-URI instructs the browser to launch a channel. In this case, the browser does not show other channels associated with the image. Consider a malicious channel that associates itself with the same image as a benign channel. If the user previews the malicious channel before flipping to the benign channel, the first

(a) (b) (c) (d)

Figure 3.13: Different combinations of the Junaio mascot and the QR code launch different channels.

object loaded from the malicious channel can reload the entire channel using *layar://[channelname]* and the other, benign channel will no longer be visible to the user.

The other risk is composite images that include a trusted image in an unexpected visual environment. When faced with a composite image, Junaio's choice of the channel to launch depends on the camera angle and distance. For example, the images in Figs. 3.13(a) and 3.13(b) launch different channels depending on whether the mascot or the QR code is more prominent. Sometimes, changing the angle of the camera by a few degrees or less changes which channel is launched. The image in Figure 3.13(c) automatically launches the channel associated with the mascot when scanned from a close distance, and the one associated with the QR code when scanned from further away. Figure 3.13(d) launches the channel associated with the QR code, even though the mascot is visible. This means that even after scanning a familiar image, a user cannot be sure that the automatically launched channel is the one he expects. Another image in the camera's field of vision may "re-route" Junaio's image recognition algorithm to a different channel.

3.7.3 How to do it right

The risk of an AR attacker registering an image trigger that is specific to an individual (e.g., a license plate) is inherent in AR services. A service may attempt to filter out such images during channel registration, but this requires deep semantic analysis of submitted images and will be inevitably bypassable. This inherent risk is exacerbated, however, by the fact that AR content is executed immediately after the image is scanned.

First, AR browsers should inform the user about the origin of AR content before launching it (at the very least, display the developer's name and basic information about the channel). Second, automatic, image-triggered code execution is fraught with danger and should be used sparingly—for example, only with trusted channels—and not with every image that happens to fall into the camera's field of vision. Third, AR browsers should develop better user interfaces that inform users about the possibility of spurious image matches and non-deterministic launches of unexpected content.

3.8 Outsourced image processing

AR browsers must continuously analyze the device's camera feed in order to recognize automatic content triggers and to anchor or position AR objects on the screen. Existing AR browsers such as Junaio and Layar do not process the captured images on the device; instead, they send them to central AR servers.

Figure 3.14: An image sent by the Layar browser over HTTP so that the Layar server can recognize content triggers. Note the accidentally captured credit card.

3.8.1 Doing it wrong

There are several reasons for outsourced image processing. First, for business reasons—injecting ads, charging content providers, keeping usage statistics, etc.—all AR content retrieval is mediated by the AR service. To facilitate image-based channel launching, recognition of trigger images is done at the server. Second, this involves matching against proprietary databases using proprietary algorithms. Centralized image processing helps protect intellectual property and removes the need to replicate and update the service's image database on millions of devices. Third, many image recognition algorithms are computationally intensive and would heavily task low-powered mobile devices.

3.8.2 Risks

In this section, we are concerned with (1) network attackers who observe network traffic between the device and the provider's AR server, and (2) the AR service itself.

Accidental overcollection of sensitive data is a big risk in this setting. For example, the Layar browser sends raw images from the device's camera to the

server over unencrypted HTTP. If such an image contains sensitive items (see Figure 3.14), they are leaked to any Wi-Fi eavesdropper.

Even if network communications were secure, the AR service inevitably collects a tremendous amount of raw visual data about its users' physical environment. This is an inherent design flaw of all existing AR services. The users must trust them to safeguard captured images, which contain a lot of sensitive information that is completely irrelevant to the AR functionality: accidentally captured computer screens, credit cards, license plates, etc. For example, in addition to the unencrypted camera image of its environment sent at the start of each scan, the Layar browser occasionally sends a log message to the server with the location of the phone. The user is not informed about this information leakage and, in general, has no way to learn which data is sent to AR servers.

3.8.3 How to do it right

When image recognition is outsourced to the server, a secure protocol should be used to prevent accidental leakage of irrelevant information in the images. If the server is attempting to recognize a channel trigger on a magazine page, there is no need for it to "see" the physical objects surrounding the page.

This is a difficult problem, but there has been some recent progress. Osadchy et al. described a prototype system for secure outsourced face matching [71]. This system cannot be directly applied in AR browsers, however, because images matched by AR browsers may appear in different lighting, under different angles, etc. Another approach is taken by Darkly [53], which can perform simple computer

vision tasks without access to raw image details.

3.9 Visual composition

To render AR content—images, text, 2D and 3D models, and HTML content from multiple origins (e.g., the channel itself and advertisers)—on top of the camera feed, AR browsers maintain a complex visual stack. For example, Junaio's visual stack is described in Section 3.6.

In Layar, a channel can use a webpage as an AR object, called an *HTML widget*. Each widget opens in its own WebView and does not display URLs or any Web-browser buttons. HTML widgets may not be covered by other types of AR objects, but can be overlaid on each other to create a visual AR experience.

3.9.1 Doing it wrong

Conventional Web browsers provide the *iframe* abstraction that allows composition and stacking of HTML content from different origins. To defend against clickjacking, a webpage can ensure that it is not framed by a page from a different origin, either via framebusting [84] (special code that moves the page to the top frame), or via X-Frame-Option [103].

AR browsers must deal with both HTML and non-HTML content, and thus resort to custom mechanisms to implement the functional equivalent of *iframe*. Consequently, standard defenses based on framebusting or X-Frame-Option no longer work. For example, as described above, Layar puts each instance of HTML content into its own WebView instance. Each instance acts like an *iframe* and can be

overlaid on other instances. Therefore, a malicious AR channel can overlay content from another origin (B) on top of its own content (A) without B being technically "framed" by A.

3.9.2 Risks

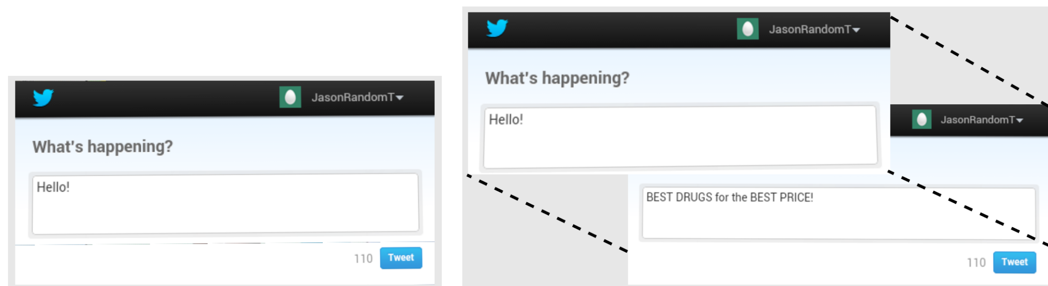
In this section, we are concerned with an AR attacker whose channel combines his own malicious HTML content with trusted HTML content from other origins.

By cleverly overlaying HTML widgets from different origins, a malicious channel can "hijack" the user's clicks. The user sees a button that appears to belong to some window, but the click is actually captured by a different window. For example, Figure 3.9.2 shows a malicious Layar channel that overlays two Twitter windows. The user may think that the visible "Tweet" button submits the "Hello!" tweet, but it actually belongs to the bottom window and thus submits the invisible, malicious tweet.

Because the victim page is in the top/main frame of its own WebView instance, it cannot prevent this attack or even detect when it is being framed in this way.

3.9.3 How to do it right

Defenses against clickjacking in AR browsers would benefit from a whole-browser equivalent of X-Frame-Option. Layar already prevents non-widget objects from covering widgets, but there is no way for HTML content to specify that its



(a) Overlapped HTML widgets in Layar. The widget with "Hello" is cut off before its Tweet button.

(b) The two HTML widgets expanded. In the attack, the bottom widget is not fully shown (its tweet text is covered and not visible to the user).

Figure 3.15: Clickjacking in Layar.

widget—or any WebView in which it is displayed—should not overlap with other widgets.

In general, using conventional embedded Web browsers such as WebView to render AR content is dangerous. WebView is not designed to display AR content, forcing AR browsers to use ad-hoc mechanisms for visually combining content from different origins. A principled solution to clickjacking in AR browsers should involve a clean-slate redesign of their user interfaces.

3.10 Indirect retrieval of AR content

AR content comes from independent third-party developers. While in theory the AR browser could fetch this content directly from the developers' servers, in practice the business models of AR services require tight control over AR content. They track usage, charge fees for channel registration, inject advertising, and, in general, monitor the interaction between their browsers and third-party content.

Consequently, content requests must pass through the AR provider's own servers.

3.10.1 Doing it wrong

Some AR browsers enable third-party channels to authenticate users or keep track of users' preferences between their visits. For example, Layar supports a cookie-based user authentication scheme that can be deployed by geolocation channels. This protocol is diagrammed in Figure 3.16.

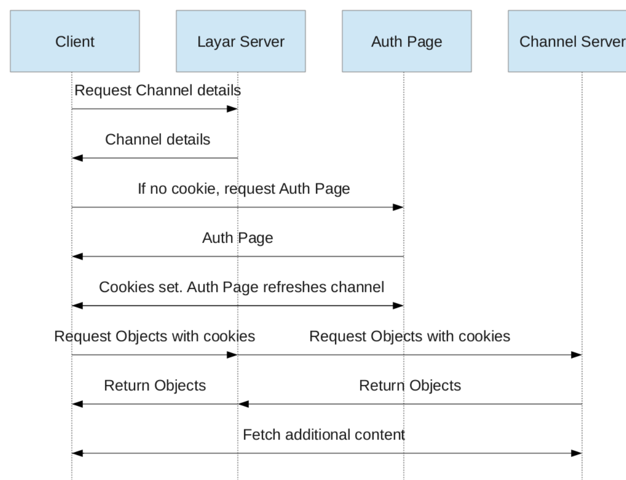


Figure 3.16: User authentication in Layar.

As mentioned above, the Layar server is involved in the launches of both authenticated and unauthenticated channels. When requesting a channel, the Layar browser sends a POST request to the Layar server and attaches the cookies associated with the channel's origin. The Layar server then attaches these cookies to the GET request it forwards to the channel server.

Cookie security depends on the binding between the cookie and its origin.

A conventional Web browser (a) maintains this binding internally, and (b) automatically attaches the cookie to every request sent to that origin. In Layar, channel launches are mediated by the Layar server, which must maintain the same cookie-origin binding as the Layar browser.

The Layar browser learns the origin of the channel from the Layar server. When the browser first loads the channel, the cookies are set by the channel's authentication page and thus correctly bound to the channel's origin at that time. If this origin changes later (e.g., the channel moves to a different domain), the Layar server notes the change and forwards all browser requests to the new location. Critically, the Layar server does *not* notify the browsers connected to the channel that the channel's origin has changed. The browsers continue to attach the cookies from the old origin to their requests, and the Layar server obliviously forwards them to the new origin.

3.10.2 Risks

In this section, we are concerned with AR attackers who lie about the URLs of their channels. By "desynchronizing" the Layar browser's and the Layar server's understanding of the channel's origin, a malicious channel can steal cookies belonging to *any* origin (Figure 3.17).

For example, the attacker initially tells Layar that the URL of his channel is *https://www.twitter.com*. When a user launches the channel, the Layar browser attaches Twitter cookies to every channel update request. Next, the attacker changes his channel's URL to *https://attacker.com*. The Layar server registers the change,

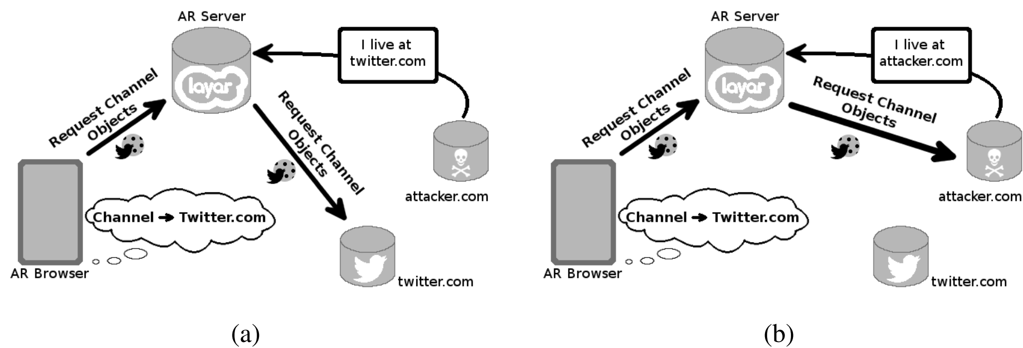


Figure 3.17: Layar cookie stealing attack.

but does not inform the browsers connected to the channel. They continue to attach Twitter cookies to every channel update request. The Layar server forwards these requests, cookies attached, to
https://attacker.com, and the attacker steals all of its users' Twitter cookies.

This attack works for any domain of the attacker's choosing (we tested it for Twitter and Facebook). Note that many AR channels are integrated with online social networks, thus the user is likely to be logged into Facebook and Twitter through his AR browser.

3.10.3 How to do it right

The first defense is to avoid replicating the state of the browser on the Layar server. The browser may request the URL of the channel server from the Layar server, but subsequent communication should be conducted directly between the browser and the channel server. The same-origin policy within the browser will then ensure that cookies are disclosed only to their origins. This defense, however,

may break Layar's business model.

The second defense is for the Layar server to ensure that it agrees with the browser about the channel server's URL. This defense requires re-engineering the protocol between the browser, Layar server, and channel server.

The final defense is to use an authentication protocol that supports delegation, e.g., OAuth. In current Layar, channels may use OAuth 1.0 to authenticate the Layar server. This protects benign AR developers from spoofed Layar servers, but not legitimate Layar servers from malicious developers, and thus does not help against the cookie-stealing attack.

3.11 Conclusions

Perceptual computing platforms help developers in creating perceptual applications efficiently. In this chapter, we presented the first in-depth analysis of the security and privacy properties of the AR browsers, one of the most popular perceptual platforms, identified multiple architectural flaws, and proposed both short-term fixes for specific vulnerabilities and some directions for future research on building secure AR browsers.

Chapter 4

Redesigning perceptual interfaces with recognizers

4.1 Introduction

Today's perceptual 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 [22, 70], or cloud object recognition services, such as Lambda Labs or IQ Engines. Existing perceptual computing platforms 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 platforms do not support fine-grained permissions required for development of least-privilege applications, relying instead on careful pre-publication vetting of applications [21].

The perceptual data coming out of the sensors often contain sensitive information. An application, however, may not need any of this sensitive information to do its job. For example, Figure 4.1 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

Figure 4.1: Perceptual 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.

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 perceptual platform; this principle of perceptual applications benefiting from "least privilege" is more general. We show two mobile phone examples in Figure 4.2. 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 4.3 shows the top 5 Amazon best-selling

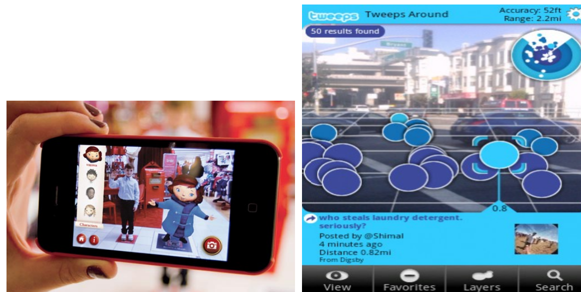


Figure 4.2: Two examples of mobile perceptual applications that only need specific objects in a sensor stream. On the left, Macy's Believe-O-Magic only needs the location in the frame of a special marker, on top of which it renders a cartoon character. On the right, Layar only needs to know the GPS location and compass position to show geo-tagged tweets.

Kinect-enabled applications for the Xbox 360, along with the Xbox Dashboard and representative perceptual apps on mobile phones. For each application, as well as the Xbox Dashboard, we enumerate the objects recognized; in Section 4.4 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 perceptual platform allows a user to restrict access at finer granularity.

The Recognizer Abstraction. To address this problem, we introduce a new least-privilege abstraction called a *recognizer* for perceptual computing platforms. 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 the perceptual computing platform, we enable least privilege for perceptual applications. We assume a fixed

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 4.3: Sample perceptual applications and the objects they recognize. Kinect apps are above the line, mobile below.

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 4.4).

Supporting recognizers in the perceptual platform 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 a perceptual platform 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.

Challenges. We faced several challenges designing our recognizer-based perceptual platform. First, other fine-grained permission systems, such as Android, have been shown to be difficult to interpret for users [37]. 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 4.7). 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 perceptual platform 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 platform 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 [70].

Our final challenge concerned recognizers that require heavyweight object recognition algorithms which may run poorly or not at all on performance-constrained mobile devices [64,66]. 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 [66].

Contributions. We make the following contributions:

- We introduce a new perceptual platform abstraction, the *recognizer*, which captures the core object recognition capabilities of perceptual applications. Our novel fine-grained permission system for recognizers enables least privilege for perceptual 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 perceptual 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 perceptual platform to offer services such as *offloading* and *cross-*

application recognizer sharing that improve performance. Our implementation has negligible overhead for single applications, yet greatly increases performance for concurrent applications and allows the platform to offload heavyweight recognizer computation.

In the rest of the chapter, Section 4.2 discusses our recognizer abstraction, and Section 4.3 describes our implementation. Section 4.4 evaluates privacy goggles, recognizers required for shipping perceptual applications, recognizer error correction, and performance of our prototype. Sections 4.5 and 4.6 present related and future work, and Section 4.7 concludes.

4.2 The recognizer abstraction

We propose a new perceptual platform abstraction called a *recognizer*. A recognizer is a component of the perceptual computing platform that takes a sensor stream, such as video or audio, and "recognizes" objects in the sensor stream. For example, Figure 4.4 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 perceptual applications operate on specific entities with high-level semantics, such as the face or the skeleton. To enable least privilege, the platform 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 perceptual platform that fire for events from a particular recognizer; the callbacks accept arguments of the specified type. For example, the recognizer in Figure 4.4 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 platform for a specific recognizer, the application must be authorized by the user. Different 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 4.4 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 platform includes recognizers for raw sensor streams, such as

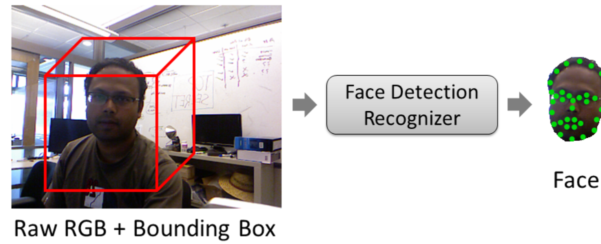


Figure 4.4: Example of a recognizer for face detection. The input is a feed of raw RGB video plus a region within that video. The recognizer outputs an event if a face is recognized in the region. Applications register callbacks that fire on the event and are called with a list of points outlining the face plus an RGB texture, but not the rest of the video frame.

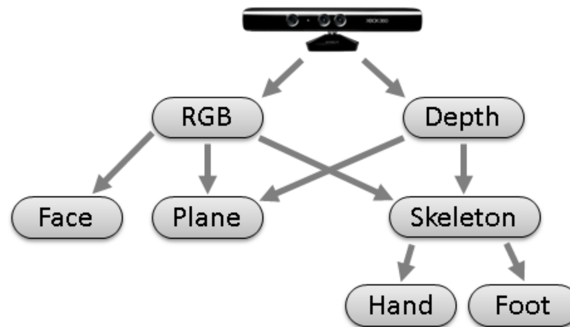


Figure 4.5: A sample directed acyclic graph of recognizers. Arrows denote how recognizers subscribe to events from other recognizers.

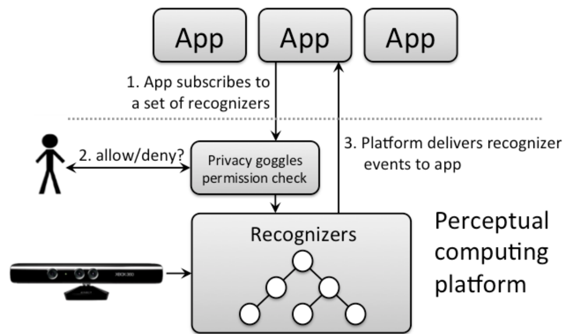


Figure 4.6: Recognizer-based perceptual platform architecture. Applications request subscriptions to sets of recognizers, which the OS then confirms with the user using privacy goggles (Figure 4.7). Once the user grants permission, the platform delivers recognizer events to subscribed applications.

RGB frames from a camera. Because subscribing to events is an explicit call to the platform, the platform can construct a *dataflow graph* showing how raw sensor streams are progressively refined into objects. Figure 4.5 shows an example. Having explicit data flow helps the platform with both security and performance, as we describe below.

Architecture and Threat Model: Figure 4.6 shows the core architecture of a perceptual platfom 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 platform confirms this request with the user using our "Privacy Goggles" visualization (Section 4.2.3). If the user agrees to the request, the platform 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 platform, 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 work.

4.2.1 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 4.4, a small set of recognizers is sufficient to cover most shipping perceptual applications.

Reducing permission requests: If an application requests access to the skeleton and hand recognizers from the DAG shown in Figure 4.5, 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 [37].

4.2.2 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 perceptual 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 [66] 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 platform preserves least privilege. For example, the platform 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.

4.2.3 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 4.7. 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 platform will compose the appropriate visualizations. In Section 4.4 we survey 462 people to demonstrate that privacy goggles do effectively communicate capabilities for "core recognizers" derived from analyzing shipping perceptual applications. Privacy goggles are complementary to existing permission widgets, such as those of Howell and Schechter [48], 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.

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 sys-

tems 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.

4.2.4 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 platform component for *recognizer error correction*. While recognizers themselves implement various tech-

niques 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 4.4.

In addition, the platform 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 platform can use the depth camera to modify the input to a face detection recognizer. By blanking out all pixels past a certain depth, the platform 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 platform 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.

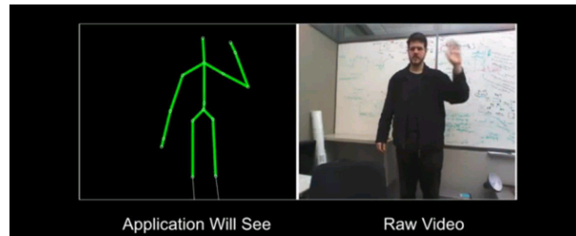


Figure 4.7: Example of "privacy goggles." The user sees the "application-eye view" for a skeleton recognizer.

4.2.5 Adding new recognizers.

Today's perceptual 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 work.

4.3 Implementation

We have built a prototype implementation of our architecture. Our prototype consists of a *multiplexer*, which plays the role of the perceptual computing platform, and *ARLib*, a library used by perceptual applications to communicate to

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

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 recognizers. When the multiplexer receives such an access request, it asks the user whether or not permission should be granted using privacy goggles (Section 4.2.3). 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 4.8. 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 4.9.

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 Kinect-Fusion (see Section 4.4.3)
FaceRecognize	RGB, FaceDetect	Name of person in current view (see Section 4.4.3)
CameraMotion	*Kinect*	Camera movements detected using an accelerometer/gyro

Figure 4.9: 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.

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 4.10 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

```
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 4.10: 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.

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 perceptual applications.

4.4 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 perceptual platform can mitigate recognizer errors 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.

4.4.1 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 4.11 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 track-

ing, 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 [81]. 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.

Privacy Expectations for Applications. To learn users' mental models of perceptual 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 4.15(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 4.2, every recognizer must implement a visualization method to enable the privacy goggles view. The platform uses these visualizations to display to the user what information is obtained by each application. We developed privacy goggles visualizations for

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 4.11: 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.

three of the four core recognizers: skeleton, hand position, and person texture. While voice commands are also a core recognizer, 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 4.13 shows an example comparing raw RGB and

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?

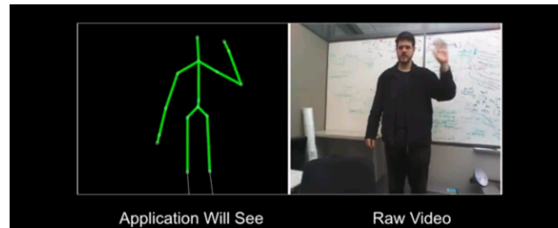


Figure 4.12: Example survey question for privacy goggles. An embedded warning video shows two views: the raw video on the right, and what the application will see on the left. Survey respondents watched the warning video, then answered questions about what the app could or could not do after installation. Out of 152 respondents, 80% correctly identified that the app could see body position, and 47% correctly determined the app could see hand positions.

face detector recognizers.

For each pair of recognizers, we asked 50 people to rate which picture contained information that was "more sensitive." Figure 4.14 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 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." [37]

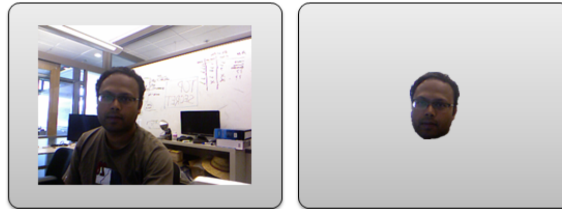


Figure 4.13: Example survey on relative sensitivity. Respondents indicated which picture is more sensitive: the "raw" RGB video frame or an image showing only the output of a face detector. Out of 50 respondents, 86% indicated the raw image was more sensitive.

We showed a short video clip of the privacy goggles visualization for the target recognizer. Figure 4.12 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 4.15 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 [99], 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% 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–

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 4.14: 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%.

60, 8.1% 60–70, and 1.8% 70 or older.

4.4.2 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 platform-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

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 4.15: 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).

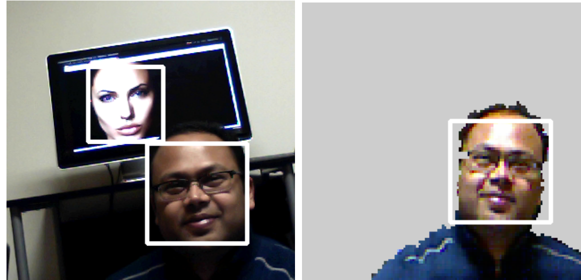


Figure 4.16: Recognizer combination in action. The left figure shows results of running a face detector on a raw RGB video frame. Two faces are detected, but only one belongs to a real person. On the right, face detection is run after combining RGB and depth. Only the real person is detected.

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.

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 number 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

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%

Table 4.1: 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.

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 4.1.

Recognizer Combination. Finally, we implemented *recognizer combination*, in which the platform 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 4.16. 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.

4.4.3 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.

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

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

Figure 4.17 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.

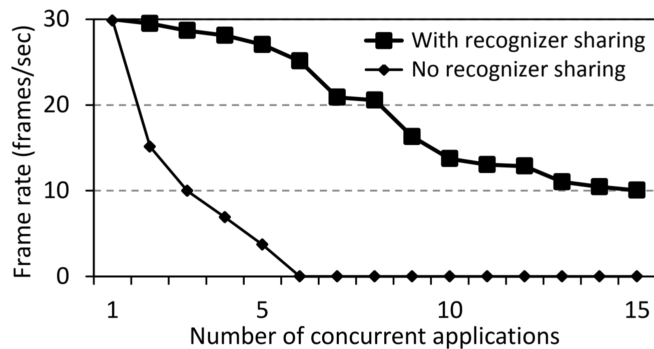


Figure 4.18: Effect of sharing a concurrent RGB video stream between applications. Our framework enables 25 frames per second or higher for up to six applications, while without sharing the frame rate drops.

Figure 4.18 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 [58]. 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

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

Figure 4.19: 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.

name to the face of someone walking by.

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 [66]. The face recognizer uses the Microsoft Face SDK [64] 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 4.19 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 4.19 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.

4.5 Related work

Augmented Reality. Azuma surveyed augmented reality, defining it as real-time registration of 3-D overlays on the real world [5], later broadening it to include audio and other senses [6]. 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 [78]. Previous work by our group has laid out a case for adding OS support for augmented reality

applications and highlighted key challenges [24].

Common shipping object recognition algorithms include skeleton detection [91], face and headpose detection [64, 100], and speech recognition [65]. More recently, Poh *et al.* showed that heart rate can be extracted from RGB video [77]. 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 [37] 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) [81] 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 4.4.1).

Another form of access control is to reduce the sensitivity of private data (e.g., GPS coordinates) available to applications. MockDroid [10] and AppFence [47] allow using fake sensor data. Krumm [57] surveys methods of reducing sensitive information conveyed by location readings. Differential privacy [29] 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 [53] 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 [31] and AppFence [47].

In access visualization, sensor-access widgets [48] were proposed to reside within an application's display with an animation to show sensor data being collected by the application. Darkly [53] 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 [18]'s notion of Con-

textual 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 [45] 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.

4.6 Future work

In this section, we enumerate some of the interesting future research directions for enhancing our recognizer-based perceptual platform.

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 described in this chapter 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 [25]. 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.

Bystander Privacy. Our focus is on protecting a user's privacy against untrusted applications. Mobile perceptual 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.

4.7 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 a perceptual platform 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 perceptual 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 platform-level offloading of heavyweight recognizer computation, and improves performance for concurrent applications. In short, the recognizer abstraction improves privacy *and* performance for perceptual applications, laying the groundwork for future platform support of rich sensing and perceptual application rendering.

Chapter 5

DARKLY: retrofitting privacy protection in existing perceptual interfaces

5.1 Introduction

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.

5.2 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.

The system model of DARKLY is shown in Fig. 5.1 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

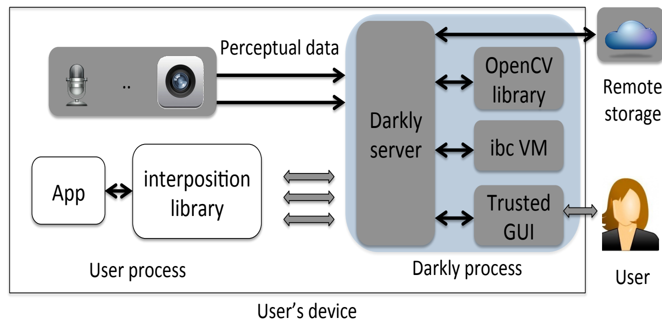


Figure 5.1: System architecture of DARKLY.

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 with-

out 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
...
```

Program 5.1: Outline of the ball-tracking robot application.

To illustrate how DARKLY works on a concrete example, Listing 5.2 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.

5.3 Privacy risks of perceptual applications

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." [16]

Many applications in DARKLY work exclusively on opaque references (Section 5.6.2), 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 5.8). 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 5.7), 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. 5.2 and 5.3). 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 [28, 30] 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 5.7.2, 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.

5.4 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 5.9) 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.

5.5 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. 5.1, 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 5.9, 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 [62], 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 5.9).

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 5.7). 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 5.8).

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 [98] to match a face against a database (see Section 5.6.6).

For such applications, DARKLY provides a special `ibc` language inspired by GNU `bc` [9]. 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 5.6.6). Furthermore, DARKLY restricts their output to 32 bits, thus blocking high-bandwidth covert channels.

5.6 Implementation

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

5.6.1 OpenCV

OpenCV provides C, C++, and Python interfaces [70] 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 5.9 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

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

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:

```
1  IplImage* img = 0;
2  // load an image
3  img=cvLoadImage(argv[1]);
4  // create a window
5  cvNamedWindow("mainWin", CV_WINDOW_AUTOSIZE);
6  cvMoveWindow("mainWin", 100, 100);
7  // invert the image
8  cvNot(img, img);
9  // show the image
10 cvShowImage("mainWin", img );
11 // wait for a key
12 cvWaitKey(0);
13 // release the image
14 cvReleaseImage(&img );
```

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

5.6.2 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 meta-data 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);
/* 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.

5.6.3 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.

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

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.

5.6.4 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).

Declassifiers apply an appropriate privacy transform (see Section 5.7) to the input, as shown in Table 5.1. For example, *cvGetImageContent* returns a thresholded binary representation of the actual image. Furthermore, these outputs are

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

Table 5.1: Transforms used for each DARKLY declassifier.

displayed on the DARKLY console to inform the user.

5.6.5 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 5.8).

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 [46]. 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); ... } }
}
```

Program 5.2: 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.

5.6.6 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 [9]. 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]

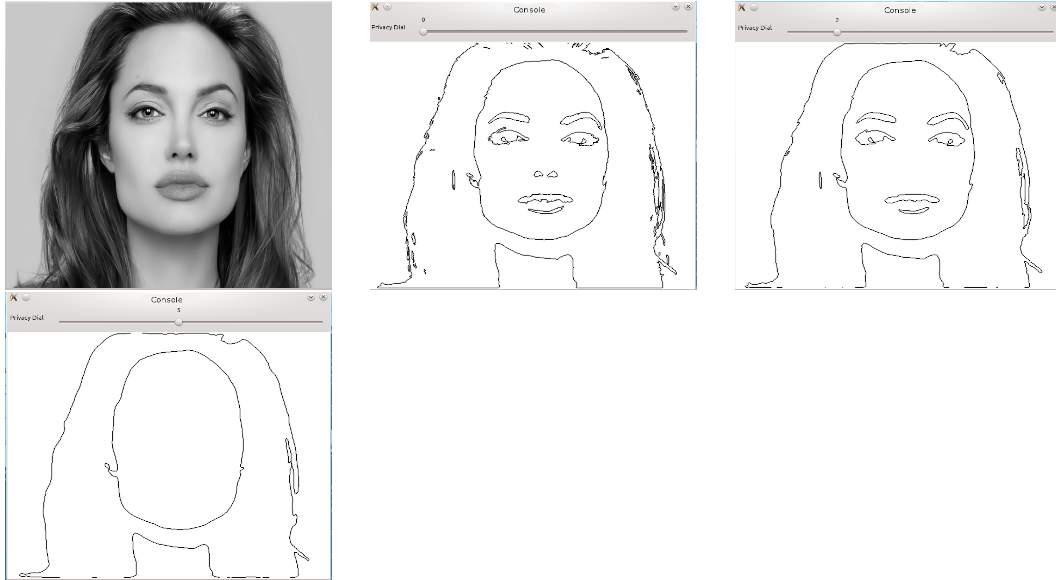


Figure 5.2: Output of the sketching transform on a female face image at different privacy levels.

`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

[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 5.3: Output of the sketching transform on a credit card image at different privacy levels.

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];
```

Program 5.3: 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);
```

Program 5.4: 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 5.9) 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);
```

Program 5.5: 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,
            &(*( 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);
```

Program 5.6: Modified face-recognition application code using `ibc` for calculating the closest match to the input image.

5.7 Privacy transforms

In Section 5.9, 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 5.6.4).

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 5.8). 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. 5.7). 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 5.7.1 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.

5.7.1 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. 5.3, 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 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 [50]. 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 non-differentiable 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.

[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.

Fig. 5.4 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. 5.5 shows that, for reasonable values of the privacy dial (3–6), the resulting sketches can be represented in 500-800 bytes.

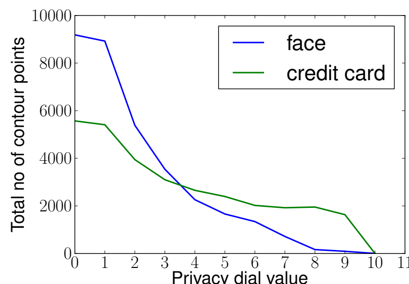


Figure 5.4: Sketching: reduction in information available to the application for images from Figs. 5.2 and 5.3.

5.7.2 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

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

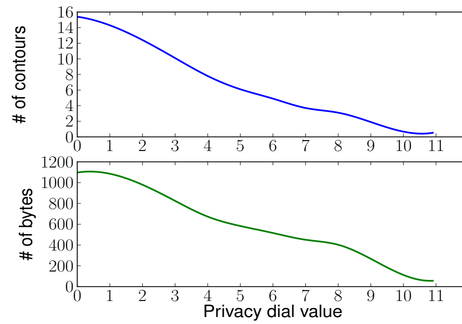


Figure 5.5: Sketching: reduction in average information available to the application for facial images in FERET database (size roughly 220x220).

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.

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 [98] that uses Principal Component Analysis to calculate a set of 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 [15]. Fig. 5.6 shows an example from a cluster of size 2 obtained by hierarchical clustering on a 40-person ORL dataset [87]. Clustering and morphing are done once to produce a fixed dic-

tionary 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.

Our *cluster–morph* algorithm is inspired in part by Newton et al.'s algorithm for k-anonymity-based facial de-identification [67], which works as follows: given a database of images, repeatedly pick a yet-unclustered image from the database

[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.

Figure 5.6: Face morphing for generalization. The left and right faces belong to the same cluster; the morph "representing" this cluster is in the center.

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.

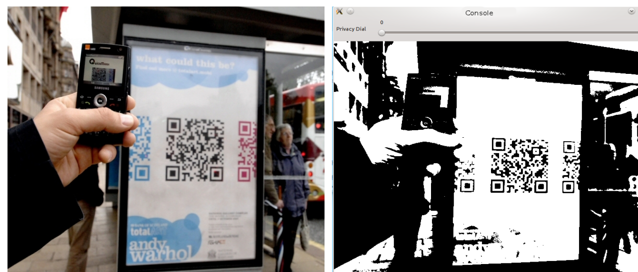


Figure 5.7: Output of the thresholding binary transform on an image of a street scene with a QR code. QR decoding application works correctly with the transformed image.

5.8 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. 5.8 and 5.9. 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.

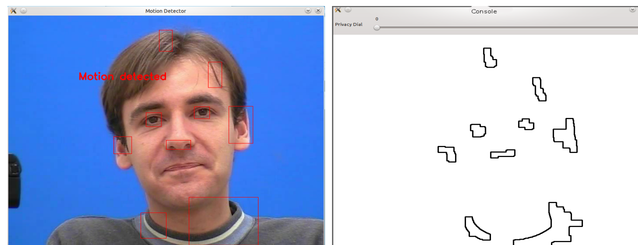


Figure 5.8: Motion detector: actual image and the DARKLY console view. Application works correctly with the transformed image.

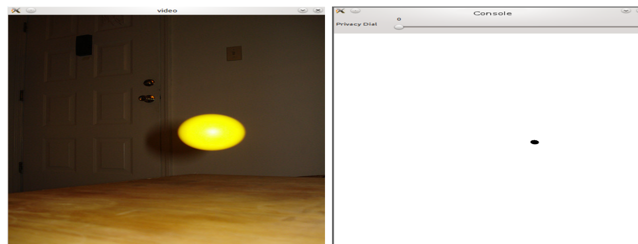


Figure 5.9: Ball tracker: actual image and the DARKLY console view. Application works correctly with the transformed image.

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 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. 5.3).

5.9 Evaluation

We evaluated DARKLY on 20 OpenCV applications, listed in Table 5.2 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 5.3. 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 5.6.6) 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.

Performance. Performance is critically important for perceptual applications that deal with visual data. If the overhead of privacy protection caused frame rates to

[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	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 backprojector	`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 5.2: Benchmark OpenCV applications.

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. 5.12	Contours
Facial features detector	258	0	No change **	Rectangular bounding boxes
Square detector	238	0	See Fig. 5.12	Contours
Ellipse fitter	134	0	See Fig. 5.12	Contours
Intensity/contrast changer for images/histograms	127	0	No change	Histograms
Ball tracker	114	0	See Fig. 5.12	Moments
PrivVideo	96	0	No change	None
Morphological transformer	91	0	No change	None
H-S histogram backprojector	81	0	See Fig. 5.12	Histogram
Laplace edge detector	73	0	No change	None
RGB histogram calculator	70	0	See Fig. 5.12	Histogram
H-S histogram calculator	58	0	See Fig. 5.12	Histogram
Hand detector	48	0	No change	Yes/no
Corner finder	42	0	See Fig. 5.12	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 5.7.2).

Table 5.3: Evaluation of DARKLY on OpenCV applications.

drop too much, applications would become unusable. Figure 5.10 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. 5.11 shows that the performance variation of the security camera application

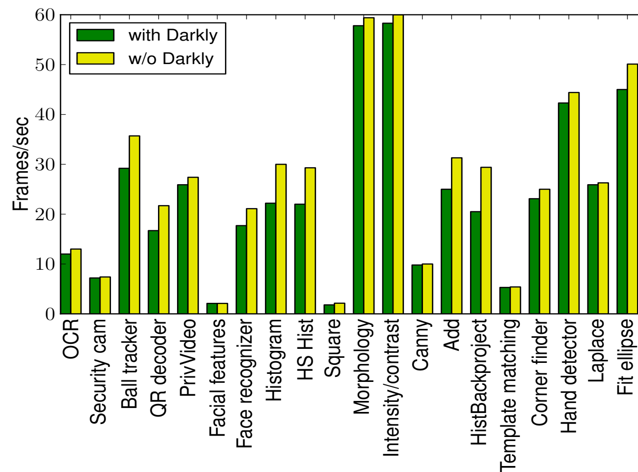


Figure 5.10: Frame rates with and without DARKLY.

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 5.3 shows that for most applications, there is *no change of functionality* and *no loss of accuracy* even at the maximum privacy setting. 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. 5.7). For the remaining applications, the tradeoff between their accuracy and user-selected privacy level is shown

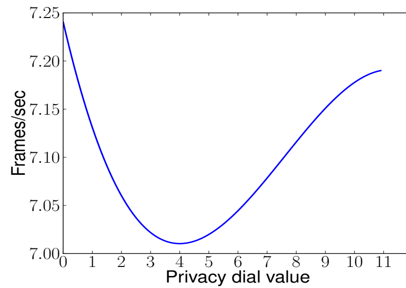


Figure 5.11: Frame rate of the security-camera application as a function of the privacy level. At levels above 4, OpenCV switches from directly calculating the convolution to a DFT-based algorithm optimized for larger kernels. Furthermore, as privacy level increases, smaller motions are not detected and the application has to process fewer motions.

in Fig. 5.12.

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.

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

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

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.

5.10 Related work

Denning et al. [26] 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 [97] 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 [71] 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 [95]. Senior et al. [90] 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 [27] 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. [17] developed a method for counting the number of pedestrians in surveillance videos without tracking any single individual.

Sweeney et al. published several papers [42, 43, 67] 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 5.7.2. 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 [48]. 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. [24] 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. [52] 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.

5.11 Conclusions

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.

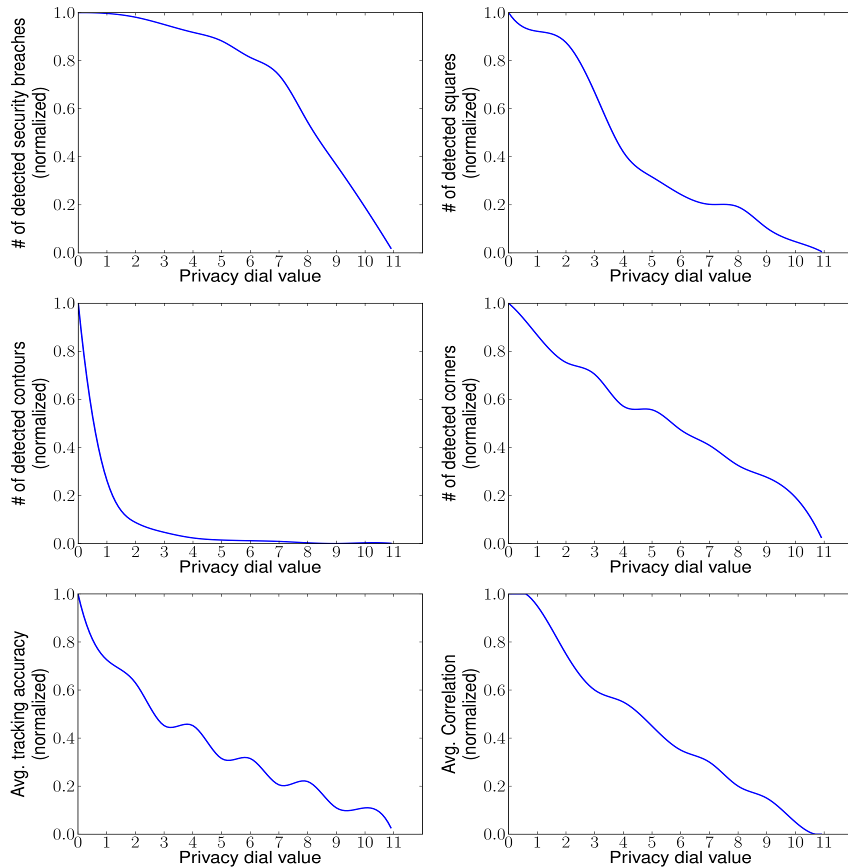


Figure 5.12: Change in the number of detected security breaches (Security cam), detected squares (Square detector), detected contours (Ellipse fitter), moments (Ball tracker), and histograms (RGB and H-S histogram calculators, Intensity/contrast changer for images/histograms, and H-S histogram backprojector) as the privacy level increases. Correlation between histograms was calculated using the cvHist-Compare function. Accuracy for tracking was measured using the Euclidean distance between the object's original position and the reported position after applying privacy transforms.

Chapter 6

Peer group analysis: automated detection of applications violating least privilege

6.1 Introduction

Software risks can be assessed in a simple, intuitive manner by clustering software from which users can expect similar functionality into *peer groups*, and by analyzing the behavior of software relative to its peer group. In this chapter we apply such *software peer group analysis* to online software markets (aka app stores), and demonstrate its effectiveness in ranking software according to risk, and in calling out risky outliers. This analysis provides clear and compact explanations for its risk-assessment results, and applies equally well to both malicious software and grayware, such as potentially-unwanted software [63]. We also show this analysis to be robust, by demonstrating that its results are meaningful, and stable, even when software is clustered into peer groups using different means, or when different aspects or metrics are used to define software behavior.

Intuitively, the security-relevant behavior of regular software is bounded by its functionality; e.g., a voice-recording application will likely need access to audio from the microphone, whereas a text editor will likely need access only to the user's files. Thus, the behavior of benign software can be circumscribed (although

this bound may change over time, e.g., as it may become common for text editors to support voice input, and for them to need access to the microphone). Meanwhile, no such bounds apply to malicious or undesirable software which may hide their malicious behavior behind a facade of innocent functionality, like a Trojan horse. Furthermore, inherently, more risk is imposed by software that is privileged to exhibit security-critical behavior, since the software's intentions cannot be certain, and may change over time, e.g., due to some compromise.

Therefore, if software exhibits behavior that is unexpected by its user, then this is a natural indicator of increased risk. Supporting such risk assessment, Section 6.3 describes the results of a user study that validates that users have different expectations of security privileges when software provides different functionality. However, such risk assessment cannot be performed in isolation: each individual user has neither the means nor the opportunity to collect, examine, and compare data about software behavior (and will be unsure of their expectations, anyway). Instead, in practice, this risk assessment must be performed centrally, e.g., by market operators, based on comprehensive data about the market's software, including crowdsourced data about software behavior and what software users find similar.

In this chapter, we describe software peer group analysis as an attractive instance of the above approach to risk assessment. We apply peer group analysis to the Chrome and Android online software markets, and describe the results in detail.For both of the markets, we found the analysis to be effective in identifying risky software as well as in ranking software to highlight for users the software with fewer privileges, relative to its peer group.

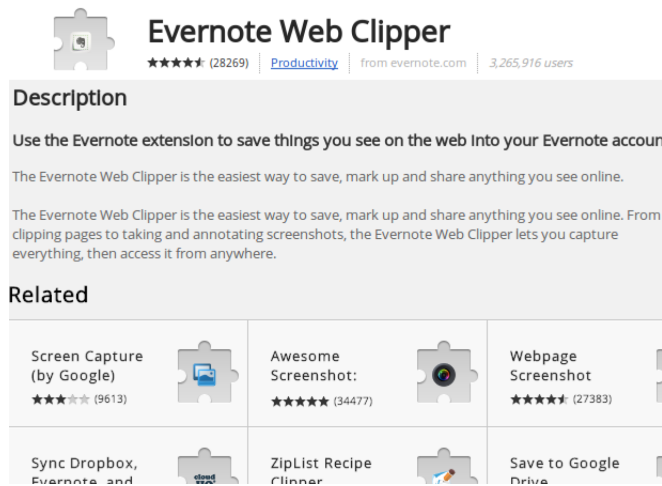


Figure 6.1: The description and related items for the Evernote software in the Chrome Web Store market.

The security challenges of online software markets. Online software markets have become a primary means by which end-user applications and other software are discovered, installed, updated, and managed. Such markets contain a great variety of software items (often in the millions), provide detailed categorization and descriptions of this software and its circumscribed behavior, and also offer extensive facilities for finding and ranking this software by either search keywords or similarity. For example, Figure 6.1 shows how the Evernote Web Clipper software appears in Google's Chrome Web Store online software market.

These markets are part of a software ecosystem that typically consists of a software platform (e.g., Google's Chrome and Android, or Apple's iOS), a community of developers writing applications for that platform, one or more online software markets (with the main one usually maintained by the platform owner), as well as enterprise administrators and other ecosystem actors. To maintain secure

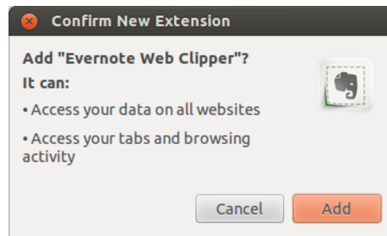


Figure 6.2: Permission screen shown while installing the Evernote extension from the Chrome Web Store.

and thriving ecosystems, platform owners and market operators must help users avoid malicious and abusive applications, and help developers adhere to the principle of least privilege [85]. All ecosystem parties stand to benefit, if a sound means of software risk assessment can be applied to help users to avoid malicious and unwanted software.

To limit software privilege, the online-software-market platforms implement *permission systems* such that application execution is isolated and circumscribed in its behavior by a developer-specified set of permissions [36]. These permissions systems allow developers to request access to platform functionality, (such as the microphone hardware), and have that access granted by some combination of the platform owners, market operators, or end-users themselves. In particular, when installing an application from the online market, users may be asked to approve a list of security-relevant privileges requested by the software, e.g., as in the graphical user-interface dialog of Figure 6.2.

Therefore, the ecosystem's security may depend on the users' ability to identify when software spuriously requests security privileges that are anomalous, given the software's purported functionality. However, this is an inherently-ambiguous

task, which users are not in a good position to perform, as hinted at by the aforementioned example of text-editor access to the microphone. Furthermore, to correctly assess the risk of such privilege requests, end users must have a good understanding of the technical details and semantics of the platform's security permissions, and how they are used by other market applications. Thus, it is not surprising that several studies have shown users to find it difficult to identify anomalous software privileges, and limit the set of security permissions granted to applications [36, 38].

Peer group analysis, ranking, and risky outliers. In this chapter, we show how a simple, intuitive, and effective means for assessing software risks can be achieved by clustering market-based software into *peer groups*, and analyzing their behavior relative to the corresponding peer groups. A peer group consists of software that (for the most part) provide similar features and functionality, and for which the user can expect similar behavior. For example, different text-editing applications are likely to belong in the same peer group, whereas mobile-phone software that provides flashlight functionality belongs in another peer group.

Software peer groups offer the most meaningful points of comparison for software whose functionality is defined by a limited, well-defined set of related features. Fortunately, most software in online software markets satisfies this property. In fact, for some markets, like the Chrome Web Store, the market operators explicitly prohibit software that bundles unrelated features, as part of their fair use policy [40].

Once peer groups have been defined, the common patterns of security-

relevant behaviors can be identified for each peer group, and software ranked in terms of how *unexpected* its behavior is compared to its peer group. Thus, the apples can be compared against the typical apple, and the oranges against the average orange. Then, outlier software whose security-related behavior is sufficiently different can be isolated, and treated specially. As we show in this chapter, such a ranking is helpful for multiple purposes like helping users to minimize risk by picking lower-risk software with the desired features, allowing market owners to identify unwanted applications, letting application developers know about potential least-privilege violation etc.

Software risk assessment is a well-established idea, as are many related forms of software analysis; recently, such techniques have been applied to the domain of online software markets [41, 72, 74]. Compared to this recent work, our main contributions are as follows.

- We introduce the notion of peer group analysis in the context of software risk assessment. We evaluate our technique and show its effectiveness on two large datasets: one with $44,000$ Chrome extensions from the Chrome Web Store and another with more than one million Android applications from the Google Play Store. We show that the results of peer group analysis is intuitive and easy-to-explain to the users. We also demonstrate that peer group analysis is robust and generic: irrespective of which technique we used to construct them, our peer groups permitted effective estimates of software risk, as long as they correctly associated software with similar functionality.

- We show that the idea of forming peer groups based on functionality is in sync with user expectations of software security privileges. With a medium-scale user study, we confirm that users expect applications to have different security privileges based on their functionality. We also confirm that, in isolation, each user has difficulty in reasoning about security permissions, by analyzing some of the complaints filed by users of software in the Google Play Store.
- We propose and evaluate three different usages of ranking risks using peer group analysis: *a*) providing users with lower-risk software for a desired feature; *b*) helping market owners to triage and identify unwanted software; and *c*) informing software developers about potential least-privilege violations in their applications.

We have developed peer group analysis in collaboration with the operators of the Chrome Web Store and Google Play Store online software markets. Our techniques have already seen limited deployment in both of those markets, where they have been used to rank, identify, and triage potentially abusive software.

6.2 Securing online software markets with peer group analysis

Peer group analysis is a common technique in business economics to compare a corporation's financial performance against its peers [104]. The absolute values of a corporation's financial performance indicators (e.g., price-to-earnings ratio [75]) depend on several external factors like the type of industry, geographical location etc. Peer group analysis identifies peers of a particular corporation that

share certain features (e.g., the geographical location and the industrial sector) and thus provides a basis to compare the corporation's financial performance against its peers.

In the context of anomaly detection, peer group analysis has been used in prior works to detect credit card frauds by Bolton et al. [12]. However, they tend to focus more on the temporal aspect of the peers' behavior. They formed peer groups by selecting the credit cards showing similar behaviors in the past and argued that their behavior should be similar in the future as well.

By contrast, in this work, we apply peer group analysis in a way that is closer to its original usage in business economics. In essence, our approach is somewhat similar to that of Keyani et al. [55] which detected attacks against peer-to-peer networks by comparing each node's estimate of the rate at which their first and second-degree neighbors are leaving the network. We identify peer groups of applications based on their functionality and compare their security privileges to get a better understanding of how atypical a particular application's privileges are compared to those of its peers. Identifying the privileges commonly shared by the applications in the same peer group allows us to indirectly estimate the set of privileges required for providing the common functionality of that peer group. Determining such mappings, without using peer group analysis, is extremely complicated and requires in-depth understanding of the platform API as well as their usage by the applications.

We define the following terms in the context of peer group analysis and use them for the rest of this chapter.

- *Peer group.* Each peer group consists of a set of software applications from which users expect similar functionality.
- *Unexpectedness.* For a given application, unexpectedness is the measure of how atypical it is relative to its peers; this unexpectedness score is assumed to be correlated to the risk that an application poses to its user.
- *Unexpected privileges.* A privilege used by an application is identified as unexpected if it is atypical relative to the privileges of the application's peers.
- *Risky applications.* An application is assumed to pose higher risk if its unexpectedness value turns out to be greater than a certain threshold. The actual value of the threshold is a tunable parameter and may vary across different peer group analysis implementations.

Peer group analysis consists of two separate steps: identifying different peer groups based on functionality and finding higher-risk applications by detecting outliers in each such peer group based on their privileges. The list of risky applications along with their unexpectedness scores can be used in several ways to make software markets more secure, as described below.

- *Helping users avoid overprivileged software.* As we show in Section 6.3, security-conscious users try to detect spurious privileges of risky applications by associating its privileges with its functionality. Unfortunately, they often make mistakes due to the complexity of the process. Unexpectedness

estimation using peer group analysis can automate this process and make it more robust.

- *Detecting unwanted software.* We confirm empirically, as described in Section 6.6.2, that high unexpectedness score is a strong indicator of the presence of potentially unwanted functionality in a software. Thus, market owners can use the unexpectedness score as a signal for detecting unwanted software.

- *Encourage developers to create low-risk software.* Automated identification of unexpected privileges can allow the market owners to provide concrete guidelines for the developers on how to lower the unexpectedness scores of their software. The market owners can also provide incentives to careless non-malicious developers to write low-risk software by ensuring that riskier software are ranked lower in the search results.

6.3 Do users expect security privileges to be tied to software functionality?

In this section, we show that users expect software to use different privileges based on their functionality. To that end, we first analyze the abuse reports filed by users about the Google Play store applications and show that users often complain about software with security privileges that the users cannot relate to the software's functionality. However, we also find that these complaints often make invalid assumptions about the links between security privileges and software functionality, which are often complicated, and always platform-dependent. Next, we describe

an Amazon Mechanical Turk (MTurk) study with 300 participants whose results confirm that users have different privilege expectations from applications providing different functionality.

6.3.1 User reports of privilege abuse

We analyzed one month's submitted abuse reports, from around the end of 2013, where users of Google Play Store software complain about abusive security permissions. The dataset contained abuse reports for over 200 different applications, with between $1 - 40$ abuse reports per application. We find that, in general, users complained that applications require way too many permissions for their corresponding functionality. Many reports also explicitly stated several mismatches between the expected functionality and the requested privilege. For instance, one such report complained about a banking application requesting permissions to take pictures or videos. Note that this user did not realize that this banking application needed the picture of a check for automatic check processing. Another report was concerned about a flashlight app requiring full network access. Similar to the previous case, the user did not understand that this application, like many other free ones, use network access for serving advertisements.

These abuse reports show that users often try to detect unexpected privileges, depending on software functionality, in line with previous findings by Jin et al [60]. However, the reports also show that users may require significant assistance in performing such analysis, e.g., by receiving a better basis for comparison from an automated system.

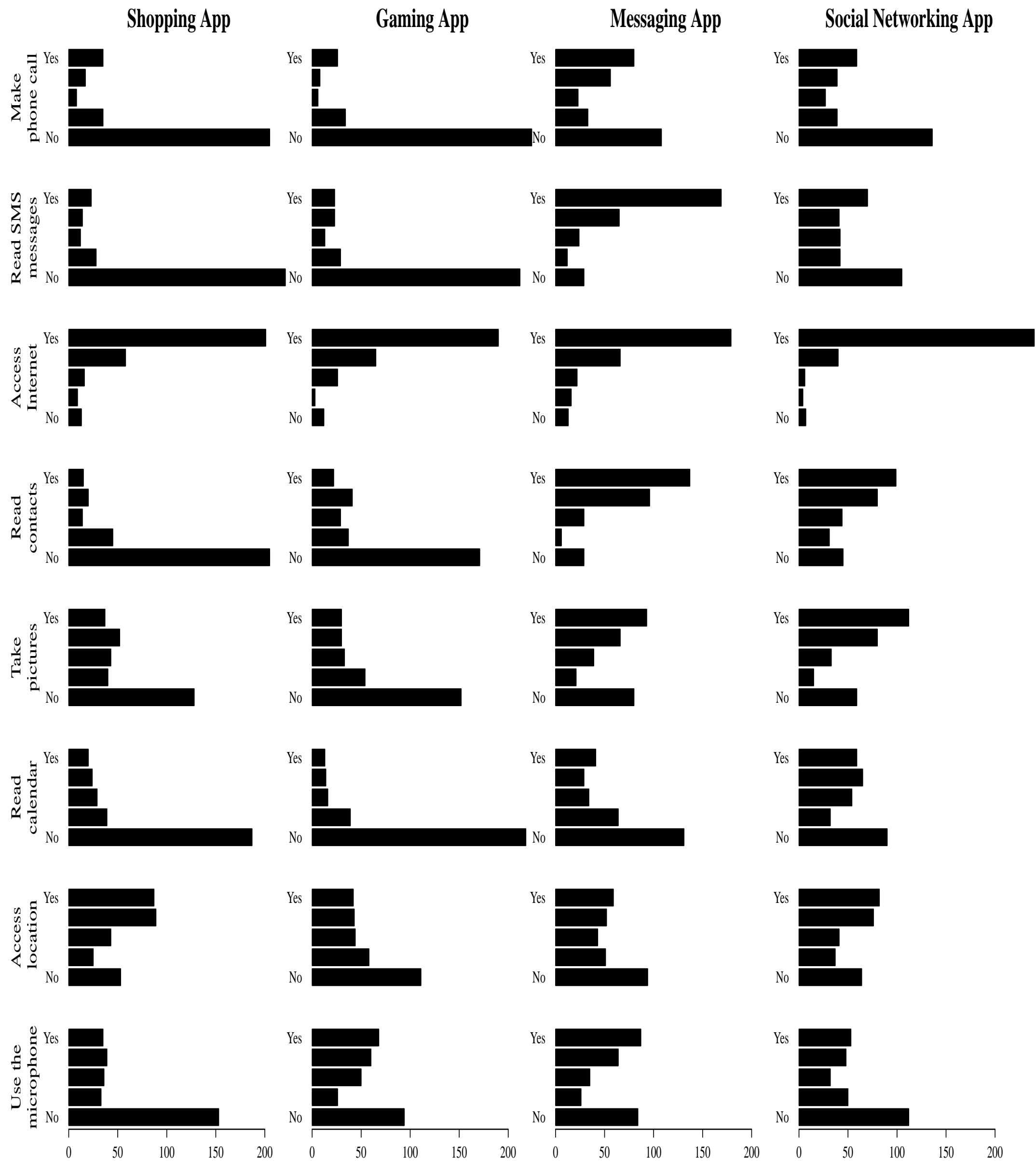


Figure 6.3: User expectations of different security privileges (making a phone call, read SMS messages, access the internet, read the contacts on the phone, take pictures, read calendar entries, find out phone's location, and use the microphone) required by applications with different functionality (social networking, shopping, gaming, messaging).

6.3.2 User study results

We conducted an MTurk survey with 300 participants to test if user expectations of the privileges required by specific applications depend on the application functionality. We had the survey reviewed by experts from our institution, and piloted with 20 people on MTurk before launching it. Participants took on average 6 minutes to complete the survey, and were compensated with 0.91 US Dollar.

Previous studies showed that the MTurk population was more diverse than that found on a typical college campus and that using MTurk could result in high-quality data [14, 73]. However, as with any online survey, participants on MTurk may cheat by answering all questions quickly without reading thoroughly to collect the compensation. To prevent such spurious data affecting our results, we restricted our survey only to the respondents who had a task approval rate of 95% or better and had completed at least 100 tasks on MTurk. Furthermore, our survey had one trap question, which had one single, obviously right answer. We ensured that all participants answered that question correctly. Finally, one member of the research team reviewed all responses to the open-ended questions to ensure that responses were on topic.

Participant demographics. Participants in our MTurk survey were skewed slightly towards male and young: 68% male, 32% female; 32% were between the ages of 18-24, 41% were 25-34, 18% were 35-44, 5% were 45-54, 3% were 55-64, and 1% were 65 or over. As to their education, just over a third of the participants had Bachelor's degrees and another third had 'some college'. The remaining third was

spread over a broad range: from 'some high school' (1%) or a Doctoral degree (1%) to having a Master's degree (8%) or a regular high school diploma (9%). Two thirds of the participants were employed full- or part-time or were self-employed (40% full-time, 15% part-time, and 13% self-employed). 17% of the participants were students, some of whom were also employed, and 16% were unemployed or looking for work. Participants represented a broad range of occupations like driver, editor, photographer, library assistant etc.

Results. Figure 6.3 summarizes the results of our user study. We asked the participants to rate on a 5 point Likert scale ('No', 'Probably not', 'Neutral', 'Probably yes', and 'Yes'), whether a *social networking app*, a *gaming app*, a *messaging app*, and a *shopping app* should be able to do a set of actions if installed on a cell phone. We intentionally did not use the worlds "permissions" or "privileges," since previous research has found that not all users know what application permissions are [38].

Our results confirm that users expect applications with certain functionality to access specific resources on the phone, but not others. Participants almost unanimously expected a shopping and a gaming application to not be able to read SMS messages or read the contacts on the phone, while they thought a messaging application should be able to have both of these privileges. However, opinions about some privileges were mixed. For example, one participant said a shopping application should be able to find out the phone's location: *"Absolutely needs location to find more relevant deals"* while another disagreed: *"The app should not be able to access GPS. It should simply ask you the first time you install for your zip code"*.

Note that both agree that shopping applications can have access to some form of location information, but they differ in how fine-grained they think it should be.

6.4 Estimating unexpectedness using peer group analysis

In this section, first we show how software market peer groups can be identified using several different sources of information that most markets already maintain about their hosted software. Next, we enumerate different ways for approximating the security-relevant behaviors of an application. Finally, we describe how one can compute an application's unexpectedness score.

6.4.1 Identifying peer groups

As peer groups are based on application functionality, in order to identify the peer groups, we must first be able to infer different applications' functionalities. Fortunately, most existing software markets maintain different sources of information about an application's functionality: classification into pre-defined static categories, list of other related applications, textual descriptions, screenshots etc. The markets usually maintain such information to help users in finding out alternative applications providing similar functionality. We can simply leverage these existing sources of information to classify applications into different peer groups as described below.

Classification provided by the developers. Most markets including the Chrome Web Store and Google Play Store require the developers to classify their applications in one out of a fixed set of pre-defined categories while registering them

Figure 6.4: Word cloud showing the words taken from the 'video download helper' extension's textual description hosted in the Chrome Web Store. The font sizes of the words are proportional to their frequencies.

with the markets. These categories are broad and each of them cover a large set of functionality. For example, Chrome Web Store supports different categories for applications like shopping, sports, news, blogging etc. We can create peer groups of applications using categories by simply putting all applications belonging to each category in one peer group.

However, if category information is used to form peer groups, a malicious developer may try to willingly mis-categorize her application to avoid it from being detected as a high-risk application. However, we expect the security conscious users to detect such cases and report them to the market owner. In fact, while analyzing the abuse reports submitted to the Google Play Store as described in Section 6.3.1, we found several such cases .

Classification using application metadata. Most software markets contain a large amount of unstructured metadata about each application like its textual description or user comments/reviews. These sources often contain a large amount of informa-

tion about application functionality. Different automated classifiers can be designed to assign the applications to different peer groups based on these sources. For the rest of this subsection, we focus on identifying peer groups based on the textual descriptions of applications even though some of our techniques may also work on other sources like user reviews or comments.

In both the markets that we studied, all applications come with short textual descriptions about their functionality. These textual descriptions are designed to make human users better understand application features. For example, Figure 6.4 shows a word cloud representation of the words from the 'video download helper' extension's textual description hosted in the Chrome Web Store.

We explore two different techniques for designing classifiers that can use these textual descriptions to identify applications with similar functionality: one uses supervised learning algorithms to classify the applications into pre-defined categories and the other leverages unsupervised learning algorithms to cluster similar applications together. We describe both of these methods in detail below.

Supervised classification. Supervised classifiers can assign applications automatically to a set of pre-defined categories using features extracted from the textual descriptions of the applications. However, such classifiers require a set of correctly labeled applications to train on, before they can be used for classification. One way of creating such training data can be to simply use a small subset of the developer-provided categorizations from the market. Obviously, to ensure the quality of the training data, the categorizations should be manually checked.

As a proof of concept, we built a Naive Bayes text classifier and used it to classify the extensions hosted in the Chrome Web Store. We gathered 15000 extensions from the Chrome Web Store along with their textual descriptions and their developer-provided categorization in one of 19 different categories. We further divided the dataset into two sets: a training set with 9000 extensions and a test set with 6000 extensions. Our Naive Bayes classifier is trained on the descriptions and categories of the extensions in the training set. Once trained, we use the classifier on the descriptions in the test set and evaluate the accuracy of the classifier by comparing the assigned categories with the developer-provided ones. In our tests, we found that our Naive Bayes classifier was able to correctly categorize almost 60% of the extensions. While manually inspecting some of the applications that were assigned to different categories than their developer-provided ones, we found that our classifier actually categorized several extensions correctly that were mis-categorized by the developers. For example, the Google voice extension that can be used to make calls or send SMS was mis-categorized by the developer under the 'blogging' category but the our classifier correctly assigned it to the 'phone-and-sms' category.

Unsupervised classification. Unlike supervised techniques, the unsupervised ones do not need any training data and thus they can be used to detect applications with similar functionality without any manual effort.

We built a prototype implementation using Latent Dirichlet Allocation (LDA) [11] to both automatically find topics out of the textual descriptions and to estimate the likelihood of an application belonging to a topic. We classify the applications into

Figure 6.5: Word cloud showing top 20 words of a topic detected by LDA that consists mostly of games. The font sizes of the words are proportional to their probability of selection under the topic.

Figure 6.6: Word cloud showing top 20 words of a topic detected by LDA that consists mostly of 'movies' and 'videos'. The font sizes of the words are proportional to their probability of selection under the topic.

different peer groups by creating one peer group for each such topic and including all applications whose probability of belonging to that topic is higher than a threshold. In order to evaluate our prototype, we collected the English textual descriptions of $44,000$ extensions from Chrome Web Store. We used standard natural language pre-processing techniques like removing the stop words and lemmatizing to clean up the descriptions. We also removed very short words with less than 3 characters and words that do not appear in a standard English dictionary. Our implementation used the gensim library [79] for topic modeling to create a LDA model using the cleaned descriptions.

Note that for creating meaningful peer groups, the topics found by LDA must map to real application functionality. In our experiments, we found that several topics detected by LDA map to meaningful functionality. For example, Figure 6.5 and 6.6 show the word cloud representations of two of the topics found by LDA. Our findings are in line with those of Gorla et al. [41], who explored the use of LDA on the textual descriptions of Android applications for checking their API usage against the descriptions.

However, we also find that some topics generated by LDA did not relate to any particular functionality. For example, one such topic contained the following words in decreasing order of probability of their selection - ‘page’, ‘click’, ‘search’, ‘link’, ‘image’, ‘test’, ‘button’, ‘icon’, ‘tab’, and ‘site’. Furthermore, when we looked at some of the developer-assigned categories of the extensions belonging to this topic, we found that 46%, 21%, 19%, 7%, and 7% of them are classified by the developers under ‘productivity’, ‘communication’, ‘web development’, ‘fun’ and ‘search-tools’ categories respectively. As our primary goal is to find peer groups based on functionality, we chose not to use the LDA-based classifiers further in our experiments for the rest of this chapter.

Classification based on application recommendations. Software markets usually maintain recommendation systems to help the users in finding new applications. Most modern recommendation systems output a list of related-items for the item an user is interested in. These related-items are usually computed based on collaborative filtering i.e. by extracting patterns from different users’ behavior. For example, the Chrome Web Store displays a list of related applications when a user looks at

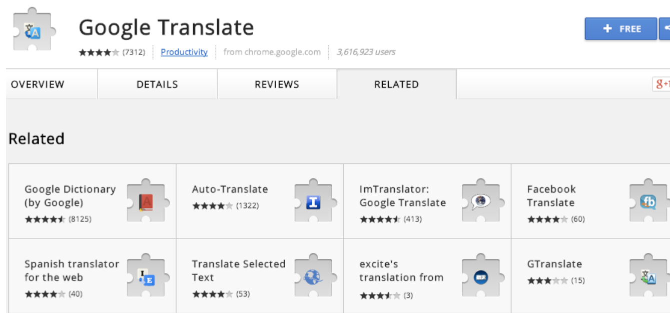


Figure 6.7: The related software items for the Google Translate extension in the ChromeMarket.

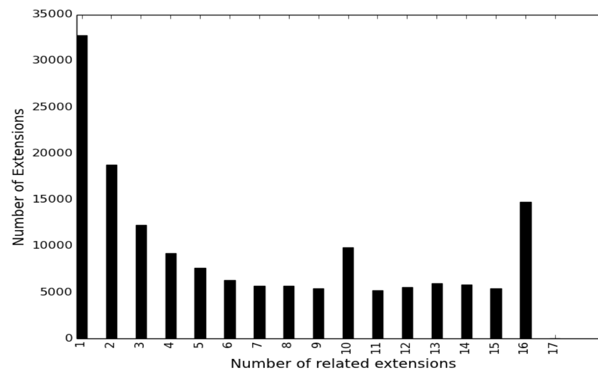


Figure 6.8: Distribution of the number of related extensions extracted for each extension hosted in Chrome Web Store.

the details of any particular application. Figure 6.7 shows the list of related applications for the Google Translate extension. One can clearly see that the related applications provide similar functionality i.e. translate text from one language to another.

One simple way to create the peer group from the related application list is to put all related applications in one group. Figure 6.8 shows that this technique is very effective in creating small tightly-knit peer groups.

6.4.2 Estimating security-relevant behavior

Before we can compare an application against its peers, we must be able to enumerate all security-relevant behaviors of the application. A simple and effective way of circumscribing such behaviors is to create a list of the security privileges used by the application. Most modern software platforms including Chrome and Android use pre-defined permissions to restrict an application's privileges to access arbitrary resources. Therefore, in such systems, the set of permissions used by an application describes the privileges it used. However, finding the exact set of permissions that are actually used by an application is a hard problem. Below, we describe two different ways to approximate permissions used by applications.

Requested Permissions. In most static permission-based systems like Chrome or Android, application developers must declare all the permissions that their applications need to operate correctly. However, in most cases, the application writers tend to over-estimate the permissions used by their applications [3, 35].

Estimated Permissions. Permissions used by an application can be estimated by first collecting all API calls that an application makes, then using a platform-specific mapping between the API calls and the permissions to enumerate the corresponding permissions [3, 35]. The API calls made by an application can either be estimated statically or dynamically. The static techniques often overestimate the API calls due to the presence of unused functions while the dynamic ones underestimate the API calls because of the lack of full coverage.

6.4.3 Estimating unexpectedness

For computing unexpectedness of an application's privileges relative to its peers, one can use standard machine learning techniques like One-Class Support Vector Machines (OC-SVM) [89]. However, one of the major drawbacks of these techniques is that the rationale behind their decisions is often hard to explain to a human. We chose to use a simple intuitive technique, shown in Algorithm 1, for computing unexpectedness scores. Our technique is easy to understand and reason about. This is a necessary condition in our setting as the the unexpectedness scores are designed to be interpreted by the users.

The basic intuition behind Algorithm 1 is to first isolate the uncommon privileges for each peer group and then compute each application's unexpectedness score based on how many such uncommon privileges it uses. W_p in Algorithm 1 indicates the amount of weight we assign to each such uncommon privilege.

6.5 Experimental setup

We evaluate our techniques on two different software markets: the Chrome Web Store and the Google Play Store. We describe our experimental setup for each of these markets in detail below.

Extensions from Chrome Web Store. We collected a set of $44,000$ Chrome extensions covering all the extensions that were published in the Chrome Web Store during early 2014. For each extension, we extracted its developer-provided category, the list of related extensions, and the list of requested permissions from the

Algorithm 1: Computing the unexpectedness value of an application a with respect to peer group g.

```
for all application a in market do
  unexpectedness_a ← 0
  P_a ← privileges used by a
  g_a ← peer group of a
  for all p ∈ P_a do
    N_g ← number of applications in g_a
    N_gp ← number of applications in g_a using p
    X_gp ← N_gp/N_g
    if X_gp < relative_frequency_threshold then
      unexpectedness_a ← unexpectedness_a + W_p
    end if
  end for
end for
```

Chrome Web Store. To extract the set of requested permissions, we first parse declared permissions from the extension manifests, removed any invalid permissions that the developers may have added by mistake. As specific host permissions are usually rarely repeated across applications, we also filtered out such permissions except all_urls.

Android applications from Google Play Store. We gathered more than a million Android applications covering all applications present in the Google Play Store during a specific day in the last six months. We extracted the developer-provided category, the list of requested permissions from the manifest, and the application binary for each application. The application binaries were disassembled using the smali disassembler and analyzed to enumerate all API calls as well as the permissions required to successfully make those API calls. To estimate the permissions

from API calls, we used the mapping of API calls to permissions provided by Au et al. [3] For the rest of this chapter, we refer to these sets of permissions as "estimated permissions" of an application. Note that neither the estimated permissions nor the requested permissions exactly represent the permissions used by an application as an application binary often contains code that do not get executed and developers often request permissions that are never used by their applications. Therefore, to get a better approximation of the actual used permissions, we also computed the intersection of requested and estimated permissions for each application.

Implementation. We implemented our techniques using two MapReduce tasks: one for computing relative frequency of privileges for each peer group and the other for computing unexpectedness values for each application and detecting higher-risk applications. We implemented Algorithm 1 as part of the second MapReduce task. For all our tests, we set W_p to 1 for the privileges that we deemed security sensitive and to 0 for the rest.

6.6 Evaluation

In this section, we first use our Chrome extension and Android application datasets to evaluate how the actual methods for peer group estimation and the values of different settings for peer group analysis affect the estimated unexpectedness scores. Next, we explore how useful the unexpectedness scores are for different purposes (e.g., helping users avoid risky, overprivileged applications, helping developers adhere to the principle of least privilege, and detecting unwanted applications) using the same datasets.

6.6.1 Effects of peer group parameters

To estimate the effects of different settings of peer group analysis on its effectiveness, we measure how many applications in our datasets have no unexpected privileges relative to their respective peer groups. As these applications represent low-risk regular applications, we expect them to be the majority in our datasets. A low number of such applications will indicate that the peer groups are not well formed i.e. they contain applications with completely different functionality.

Picking relative_frequency_threshold. One of the main configuration parameters for our unexpectedness estimation algorithm (Algorithm 1 in Section 6.4.3) is $relative_frequency_threshold$. This parameter decides the minimum proportion of applications in a peer group that has to use a privilege in order to label that privilege as "expected" for that peer group. Figure 6.9 shows the variation of the percentage of applications that have at least one unexpected privilege with different values of $relative_frequency_threshold$. For all of our tests, we set $relative_frequency_threshold$ to 0.10 for Chrome extensions and 0.05 for Android applications.

Peer group sizes. Figure 6.10 shows how the percentage of Chrome extensions with no unexpected privileges varies with different peer group sizes. For peer groups that contain only $1 - 4$ peers, there are only around 60% of such extensions even for a low $relative_frequency_threshold$ of 0.10. This indicates that such small peer groups may not be very effective in estimating unexpectedness as they might mark a large number of applications as unexpected. However, with peer

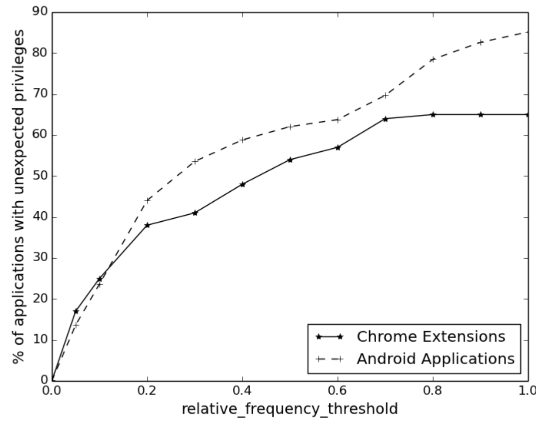


Figure 6.9: Variability of the percentage of applications with at least one unexpected privileges, for different $relative_frequency_threshold$ choices.

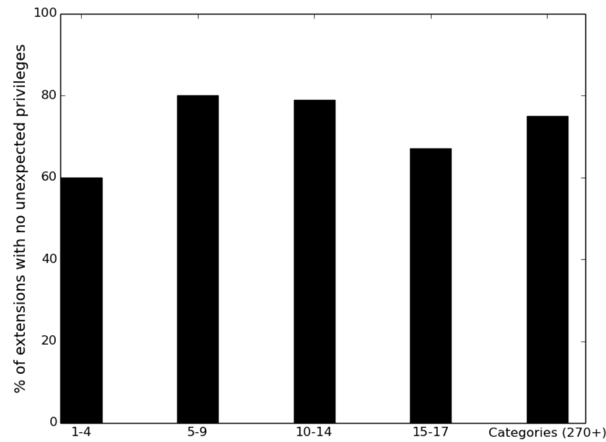


Figure 6.10: Variability of the percentage of outlying extensions with unexpected privileges, for different peer group sizes.

groups of size $10 - 14$, the percentage of extensions with no unexpected privileges rises to around 80% for the same $relative_frequency_threshold$. Nonetheless, for peer groups of sizes larger than 14, their percentage falls to around $60 - 70\%$.

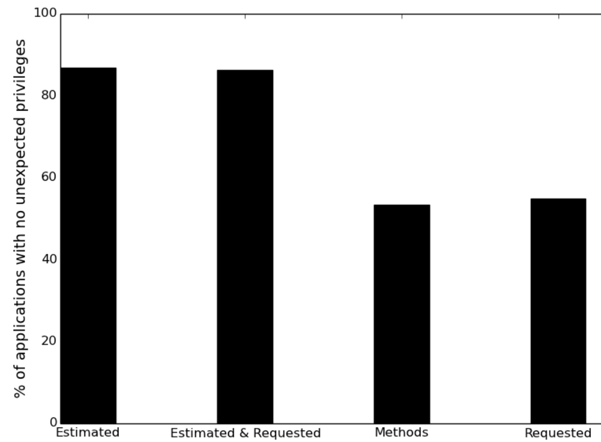


Figure 6.11: Variability of the percentage of applications with no unexpected privileges, for different privilege types.

Different types of privilege estimation. Figure 6.11 shows how the percentage of low-risk applications vary with different types privilege estimation using the Android dataset. We used four different ways for privilege estimation: requested privileges by the developers, method calls statically extracted from application binary, statically estimated permissions from application binary, and intersection of statically estimated permissions and developer-requested permissions. We found that the statically estimated permissions and the intersection of the requested and estimated permissions yield the best results.

6.6.2 Effectiveness of peer group analysis

Helping users to avoid risky applications. Unexpectedness score provides the users with a simple way to compare an application's risks with respect to its peers i.e. other applications providing similar functionality. A security-conscious user

Table 6.1: Variation of unexpectedness across the search results returned for different search queries.

Search query	App name	Score	Unexpected permissions
PDF reader	pdf reader 1	0	-
	PDF reader 2	0	-
	PDF reader 3	1	webNavigation
	PDF reader 4	3	WebNavigation , webRequest, webRequestBlocking
	Chrome-office-viewer-beta	3	clipboardWrite, fileBrowserHandler, fileSystem
Tab manager	Tab Manager 1	0	-
	Tab Manager 2	1	bookmarks
	Tab Manager 3	2	bookmarks, unlimitedStorage
	Tab Manager 4	2	bookmarks, unlimitedStorage
	Tab Manager 5	3	idle, notifications, storage
	Tab Manager 6	3	history, topsites, webNavigation
	Tab Manager 7	3	bookmarks, history, unlimitedStorage
	Tab manager 8	4	clipboardWrite, cookies, management, unlimitedStorage

should pick applications with low unexpectedness scores in each peer group.

To evaluate if it might be possible for a user to find low-risk applications that provide the desired functionality, we check the variation of the unexpectedness score across different applications providing similar functionality. In order for the unexpectedness to be helpful to the user, the scores of applications with similar features must have significant variations. To test our hypothesis, we used several different search queries like: 'pdf reader', 'tab manager', 'flashlight', 'music player' etc. and found that the unexpectedness scores of the applications shown in the search results indeed vary significantly in both Chrome Web Store and Google Play Store. Table 6.1 shows some of the results from two different queries for the Chrome Web Store: 'pdf reader' and 'tab manager'.

Encouraging developers to create low-risk software. The primary benefit of using a simple easy-to-explain method for estimating unexpectedness over sophisticated machine learning techniques is that the results can be easily explained to the de-

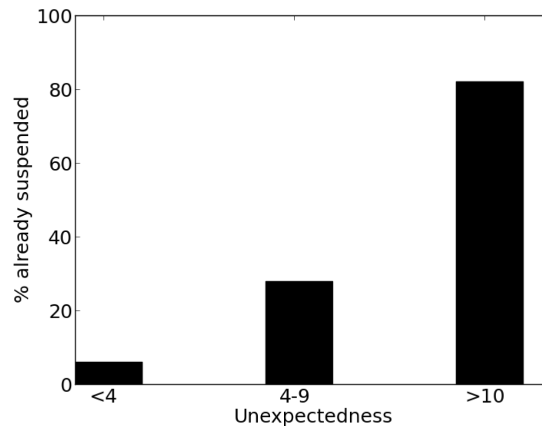


Figure 6.12: Distribution of the percentages of suspended extensions with unexpectedness score in Chrome Web Store.

velopers. This can help the non-malicious but lazy developers to either remove the offending permissions from their applications or clearly describe which features require these permissions as part of the textual descriptions of the application. For example, Table 6.1 shows that both the 'PDF reader 3' and 'Tab Manager 6' use the 'webNavigation' permission which is unexpected in their respective peer groups. With this information, the application developers can perform targeted audit of their code to check if their applications really need this permission or if there is a way to implement the desired functionality without this permission. However, if it turns out that the application indeed needs the 'webNavigation' permission for one of its features, the developer should be able to add an explain about it in the applications' textual descriptions.

Detecting unwanted applications. To evaluate our technique's effectiveness in detecting unwanted applications, we created a new dataset by picking 3828 randomly

chosen extensions from our existing Chrome Web Store dataset and augmenting them with 3828 suspended extensions. These extensions have been suspended by the market owner due to a variety of reasons including malicious activity, spamming, violating market policies etc. We compute the unexpectedness values for all extensions in the dataset including both the suspended and live extensions. For each application with a unexpectedness score greater than zero, we compute the percentage of suspended applications for different scores. Figure 6.12 show the results from the experiments using the category-based peer groups. This figure clearly demonstrate that applications with high unexpectedness scores are very likely to be suspended.

6.7 Related work

Risks of Android applications. Several prior works have explored identifying risky combinations of Android permissions to warn the users more effectively than the exiting system that shows warnings containing most of the requested permissions [32, 88]. However, identifying such combinations without considering application functionality often results in spurious warnings. To avoid such issues, machine learning algorithms have been used to analyze permission requests from different types of applications [7, 74]. Peng et al. [74] introduced the idea of risk assessment of an Android application relative to other applications and evaluated different machine learning techniques for computing an application's risk score. However, due to the complicated nature of the machine learning algorithms, such risk scores are not easily explainable to the users or the developers. By contrast,

the unexpectedness scores computed by peer group analysis are easy to understand and intuitive. Moreover, unlike Peng et al., we evaluated peer group analysis on a significantly larger dataset drawn from two different markets.

Textual descriptions and Android permissions. Gorla et al. [41] built a tool called CHABADA that clusters the textual descriptions of Android applications using LDA and identify outliers in each of the cluster according to their API usage. CHABADA can be thought of as a special case of peer group analysis that uses textual descriptions to create peer groups. We identify and evaluate several other sources for creating peer groups like related-items, categories etc. Moreover, unlike Gorla et al. whose main focus was detection of malicious software, we show that peer group analysis can also be useful for other purposes like helping users to pick safer applications or encouraging developers to write low-risk applications. Furthermore, unlike CHABADA which was evaluated only on 22500 free Android applications, we evaluated peer group analysis on a larger and more diverse dataset consisting of both free and paid applications from two different software markets: around $44,000$ from the Chrome Web Store and more than 1 million from the Google Play Store.

Pandita et al. [72] used natural language processing techniques to identify sentences in application descriptions explaining why an application needs a particular permission. Their approach is complementary to ours and can be used together to explain to the users why a high-risk application require certain permissions for its functionality.

Abusive software on Android. Prior work on the Android platform has focused on detecting malicious software by comparing against a set of characteristic features that has been extracted either manually or using supervised machine learning from known-to-be-abusive software [2, 105, 106]. Our approach is fundamentally different, as we do not learn any features of known abusers. Instead, we focus on identifying common patterns across benign applications belonging to same peer group.

User interfaces for Android permissions. Several studies have found that users often have difficulty to comprehend the Android permissions shown to them during the installation process and tend to install applications irrespective of the permissions they request [35,38]. Peer group analysis can help users in estimating the risks of installing an application without understanding each of the permissions. Roesner et al. [82] have proposed using access control gadgets provided by the operating system to allow users to grant permissions for user-controlled resources like camera in a non-intrusive and more intuitive manner. Such solutions can be used together with peer group analysis for helping users to control the sensitive privileges used by high-risk applications.

6.8 Conclusion

In this chapter, we proposed and evaluated peer group analysis for effective and easy-to-understand risk assessment of applications in online software markets. We showed that peer group analysis efficiently seeks out the high-risk applications irrespective of the actual techniques used for forming the peer groups. Our tech-

nique has already been partially deployed in the Chrome Web Store and Google Play Store software markets. We hope that our work will encourage other market owners to also adopt peer group analysis.

Chapter 7

Conclusions

In this dissertation, we showed that the emerging trend of perceptual computing presents several new and unique security and privacy challenges. We analyzed existing perceptual computing platforms and showed that none of them provide strong guarantees against such risks. Moreover, several such platforms e.g., AR browsers also suffer form implementation flaws that makes them vulnerable to a wide range of attacks. In order to solve these issues, we explored several complementary techniques that leverage perceptual interfaces for creating security and privacy-preserving perceptual computing platforms: supporting fine-grained privileges, enforcing access control, applying privacy transforms to minimize the amount of sensitive perceptual data released to the applications, etc. We built a prototype perceptual platform with support for fine-grained permissions. We further presented an automated risk assessment mechanism that can detect least privilege violations by comparing the security-relevant behaviors of applications with similar functionality. We believe that such automated detection of least privilege violations will incentivize application developers to use fine-grained privileges safely. We also developed DARKLY, a prototype perceptual platform that leverages several of the techniques described above to provide strong security and privacy guarantees.

In summary, this dissertation is the first step towards creating a perceptual computing ecosystem with strong security and privacy guarantees. We believe that perceptual computing is an exciting new area that deserves close attention of the security and privacy researchers. Now is the time to incorporate security and privacy mechanisms in the perceptual platforms as most of them are still at the early stages of their development.

Bibliography

[1] G.D. Abowd, C.G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5), 1997.

[2] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[3] K.W.Y. Au, Y.F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *CCS*, 2012.

[4] Automatic photo tagging: Facebook friendships get creepier. `http://nakedsecurity.sophos.com/2010/12/17/facebook-friendships-get-creepier/`.

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

[6] 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.

[7] D. Barrera, H.G. Kayacik, P.C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *CCS*, 2010.

[8] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

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

[10] 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.

[11] D.M. Blei, A.Y. Ng, and M.I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[12] R.J. Bolton and D.J. Hand. Unsupervised profiling methods for fraud detection. *Credit Scoring and Credit Control VII*, 2001.

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

[14] M. Buhrmester, T. Kwang, and S.D. Gosling. Amazon's Mechanical Turk A New Source of Inexpensive, Yet High-Quality, Data? *Perspectives on Psychological Science*, 6:3–5, 2011.

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

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

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

[18] 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.

[19] Qualcomm vuforia cloud recognition service. https://developer.vuforia.com/cloud-recognition-service.

[20] Catchoom image recognition api. http://catchoom.com/documentation/api/recognition.

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

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

[23] CV Dazzle: Camouflage from face detection. http://cvdazzle.com.

[24] 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. Technical report, 2013.

[25] 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`.

[26] 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.

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

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

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

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

[31] 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 *OSDI*, 2010.

[32] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, 2009.

[33] Layar launches "world's first augmented reality store". `http://eurodroid.com/2010/04/28/layar-launches-worlds-first-augmented-reality-store`, 2010.

[34] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies*, 1(4), 1997.

[35] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.

[36] A.P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. *WebApps*, 2011.

[37] 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.

[38] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[39] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks. In *NDSS*, 2014.

[40] Google, Inc. Developer Program Policies - Chrome Web Store, 2014. https://developers.google.com/chrome/web-store/program_policies.

[41] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, 2014.

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

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

[44] J. Grubert, T. Langlotz, and R. Grasset. Augmented reality browser survey. *Institute for Computer Graphics and Vision, University of Technology Graz, Technical Report*, (1101), 2011.

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

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

[47] 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.

[48] 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.

[49] L.S. Huang, A. Moshchuk, H.J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

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

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

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

[53] S. Jana, A. Narayanan, and V. Shmatikov. A scanner Darkly: Protecting user privacy from perceptual applications. In *S&P*, 2013.

[54] Become a junaio developer. `http://www.slideshare.net/metaio_AR/why-to-become-a-junaio-developer`, 2013.

[55] P. Keyani, B. Larson, and M. Senthil. Peer pressure: Distributed recovery from attacks in peer-to-peer systems. In *Web Engineering and Peer-to-Peer Computing*, pages 306–320. 2002.

[56] R. Kooper and B. MacIntyre. Browsing the real-world wide web: Maintaining awareness of virtual information in an AR information space. *International Journal of Human-Computer Interaction*, 16(3), 2003.

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

[58] Layar. Layar, 2013. `http://www.layar.com`.

[59] Layar introduction for developers. `http://www.slideshare.net/layarmobile/layar-introduction-for-developers`, 2011.

[60] J. Lin, S. Amini, J.I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *UbiComp*, 2012.

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

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

[63] Microsoft, Inc. How Microsoft antimalware products identify potentially unwanted software, 2014. `https://www.microsoft.com/security/portal/mmpc/shared/objectivecriteria.aspx`.

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

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

[66] 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.

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

[68] ObscuraCam: Secure Smart Camera. https://guardianproject.info/apps/obscuracam/.

[69] Open Geospatial Consortium. OGC augmented reality markup language 2.0 (ARML 2.0) [candidate standard]. http://www.opengeospatial.org/projects/groups/arml2.0swg, 2013.

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

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

[72] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: towards automating risk assessment of mobile applications. In *USENIX Security*, 2013.

[73] G. Paolacci, J. Chandler, and P. Ipeirotis. Running experiments on amazon mechanical turk. *Perspectives on Psychological Science*, 5:411–9, 2010.

[74] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of Android apps. In *CCS*, 2012.

[75] S.H. Penman. The articulation of price-earnings ratios and market-to-book ratios and the evaluation of growth. *Journal of Accounting Research*, 34(2), 1996.

[76] C. Perey. A proposal for AR browser interoperability. `http://www.perey.com/ARStandards/AR_Browser_Interoperability_Architecture_Jan_21_2014_v1_2.pdf`, 2014.

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

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

[79] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010.

[80] F. Roesner, T. Kohno, and D. Molnar. Security and privacy for augmented reality systems. In *Communications of the ACM*, 2014.

[81] 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.

[82] 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 *S&P*, 2012.

[83] G. Rydstedt, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In *WOOT*, 2010.

[84] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: A study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[85] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[86] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

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

[88] B.P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *SACMAT*, 2012.

[89] B. Schölkopf, J.C. Platt, J. Shawe-Taylor, A.J. Smola, and R.C. Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[90] 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.

[91] 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.

[92] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[93] Same origin policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[94] J.C. Spohrer. Information in places. *IBM Systems Journal*, 38(4):602–628, 1999.

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

[96] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

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

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

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

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

[101] Wikitude for agencies. `http://www.slideshare.net/wikitude/wikitude-media-portfolio-presentation`, 2012.

[102] Z. Wu, Q. Ke, M. Isard, and J. Sun. Bundling features for large scale partial-duplicate web image search. In *Computer Vision and Pattern Recognition*, 2009.

[103] The X-Frame-Options response header. `https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options`.

[104] Q.J. Yeh. The application of data envelopment analysis in conjunction with financial ratios for bank performance evaluation. *Journal of the Operational Research Society*, 1996.

[105] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *S&P*, 2012.

[106] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

Vita

Suman Jana is a Ph.D. student in the Department of Computer Science at the University of Texas at Austin. Suman was born in Paniparul, West Bengal, India. He earned his bachelor's degree in 2003 from Jadavpur University, with major in Computer Science. He completed his masters in Computer Science from University of Utah, Salt Lake City in 2009.

Email address: *suman@cs.utexas.edu*

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.