Security and Privacy from Untrusted Applications in Modern and Emerging Client Platforms

Franziska Roesner

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Security and Privacy from Untrusted Applications
in Modern and Emerging Client Platforms

Franziska Roesner

Chair of the Supervisory Committee:
Associate Professor Tadayoshi Kohno
Computer Science and Engineering

Today's computer users have the choice among an ever increasing variety of devices and platforms, including desktops, tablets, smartphones, and web browsers. In addition to these more traditional platforms, continuous sensing platforms have recently become commercially available, including augmented reality systems like Google Glass and in-home platforms like Xbox Kinect. Beyond built-in functionality, these platforms generally allow users to install and/or run applications of their choice. While these untrusted applications can provide great benefits to users, their presence also raises security and privacy risks. For example, users may accidentally install malware that secretly sends costly premium SMS messages, and even legitimate applications commonly leak or misuse users' private data.

This dissertation identifies and characterizes two significant security and privacy challenges due to untrusted applications on modern client platforms. First, these platforms often allow applications to embed content from other applications, resulting in security and privacy implications for both the parent application and the embedded third-party content. Second, these platforms ask users to make decisions about which permissions an application should receive, including the ability to access the camera or the file system; unfortunately, existing permission granting approaches are neither sufficiently usable nor secure.

This dissertation considers and tackles these two challenges — embedded third-party content and application permission granting — in the context of several modern and emerging client platforms. On the web, we study how embedded third-party content allows web applications to track users' browsing behaviors; our comprehensive measurement study informs our design of new tools and defenses. In modern operating systems (such as smartphones), we study how to secure embedded user interfaces in general, and then leverage these security properties to enable user-driven access control, a novel permission granting approach by which the operating system captures a user's permission granting intent from the way he or she naturally interacts with any application. Finally, for emerging continuous sensing platforms (such as augmented reality systems) in which explicit user input is sparse and applications rely on long-term access to sensor data, we develop world-driven access control to allow real-world objects to explicitly specify access policies, relieving the user's permission granting burden. Together, these works provide a foundation for addressing two fundamental challenges in computer security and privacy for modern and emerging client platforms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation is the culmination of several years of work and the guidance, support, and collaboration of many wonderful people.

First, I am incredibly grateful to my advisor, Tadayoshi Kohno, for his guidance, advice, support, encouragement, and friendship throughout my graduate career. Yoshi has been my greatest champion, and he has trained me in all aspects of an academic career. I have benefited greatly from his positivity and enthusiasm, and from his willingness to always provide guidance when I needed it or let me wander off in my own directions when I needed that. Thanks for showing me the ropes; I hope my subsequent career will make you proud.

I am also grateful to the other members of my dissertation committee for their guidance and support: to James Fogarty, for broadening my perspective; to Hank Levy, for his advice and mentorship on my job search and the bigger picture; and to Andy Ko, for his valuable feedback. Thank you also to all of my collaborators on the various parts of this dissertation: Crispin Cowan, James Fogarty, Tadayoshi Kohno, David Molnar, Alex Moshchuk, Bryan Parno, Chris Rovillos, Alisha Saxena, Helen Wang, and David Wetherall. I'm especially grateful to Helen Wang for her mentorship and friendship. Many others also provided valuable feedback and discussions along the way; it's been a tremendous privilege working with and learning from all of you.

I would like to thank past and current members of the UW CSE Security Lab for their collaboration, feedback, advice, insights, humor, and friendship: Alexei Czeskis, Tamara Denning, Iva Dermendjieva, Miro Enev, Karl Koscher, Adam Lerner, Amit Levy, Peter Ney, Temitope Oluwafemi, Alisha Saxena, Ian Smith, Alex Takakuwa, and Paul Vines. I'm also grateful to many other past and current members of the UW CSE department for their feedback, advice, and fruitful discussions, especially Tom Anderson, Luis Ceze, Ray Cheng, Gabe Cohn, Mike Ernst, Roxana Geambasu, Steve Gribble, Sidhant Gupta, Dan

Halperin, Ed Lazowska, and Will Scott. Thank you to Lindsay Michimoto for general grad school guidance and to Melody Kadenko and Lisa Merlin for their invaluable help with all administrative things. Thanks to the entire UW CSE community for making this place my home.

Beyond UW CSE, I am especially indebted to Doug Burger for his continued mentorship, which began during my undergraduate years at UT Austin. Thanks for teaching me how to do research and for continually challenging me to stretch myself; I wouldn't be here without you. I would also like to thank Stefan Savage for his valuable feedback and advice during my job search, Dirk Balfanz for a valuable and interesting internship at Google, and Vitaly Shmatikov for first introducing me to and exciting me about computer security and privacy.

Looking back farther, I'd like to thank a few outstanding early teachers who helped me see myself as and then become a scientist: Amanda Allen, for teaching me to love learning in elementary school; Walter Kelly, for running a science club during middle school lunch; Stan Roux, for letting a bunch of middle schoolers run experiments in his biology lab at UT Austin; and Mike Scott, for hooking me on computer science in my first semester of college at UT and encouraging me to apply for the honors program. Thanks to Carl Sagan for writing the book "Contact" with a female scientist protagonist, and thanks to my dad for taking me to see the movie in 1996; it took me many years to realize how much that fictional story must have helped shape my path.

I'm grateful to all of the friends who have enriched my life throughout grad school, both within and beyond UW CSE. Thank you especially to Nicki Dell and Nell O'Rourke, for first helping me make it through algorithms and then through the rest of grad school: you help me balance, and I can always count on you. I'd also like to thank Artemis Ailianou, Yonatan Bisk, Eric Bluestein, Amanda Cuellar, Eugene Chang, Iva Dermendjieva, Yaroslav Dvinov, Tony Fader, Mark Golazeski, Dan Halperin, Pete Hornyack, Cliff Johnson, Brandon Lucia, Nodira Khoussainova, Brenna Nowak, Kristen Reilly, Todd Schiller, and everyone else I've inadvertently forgotten to list. Thanks for amazing trips around the world, dinner parties, brunches, games nights, girls nights, ski days, happy hours; and for your unwavering support throughout my major life events, grad school struggles and successes, and job search.

Thank you to my parents, Ute and Wolfgang Roesner, for their love, support, and encouragement throughout my entire life and education. They are fantastic role models who have taught me to embrace challenges and seek balance. Thank you to my siblings, Jessi and Kevin Roesner, who are also my friends: for their support, perspectives, humor, and for giving me a home wherever they are. Thank you to my parents-in-law, Stefanie and Gus Akselrod, for their love and support. Thanks also to my cats, Tony and Fidget, for their (nearly) unconditional affection. And finally, a special thank you to my husband, Greg Akselrod, for supporting me — and let's face it, putting up with me — throughout my graduate career, and for accompanying me on many adventures. This is our part of the timelapse.

My graduate career and the work described in this dissertation were supported in part by a Microsoft Research PhD Fellowship, a National Science Foundation (NSF) Graduate Research Fellowship under Grant DGE-0718124, and a Marilyn Fries Endowed Regental Fellowship; as well as by NSF Awards CNS-0722000, CNS-0846065, CNS-0917341, and IIS-1053868, by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-12-2-0107, by an Alfred P. Sloan Research Fellowship, and by a gift from Google.

DEDICATION

To my parents, Ute and Wolfgang Roesner, and to my husband, Greg Akselrod, for their continued support of my education and career.

Chapter 1 INTRODUCTION

Today's computer users have the choice among an ever increasing variety of devices and platforms, including desktops, tablets, smartphones, and web browsers. In addition to these more traditional platforms, continuous sensing platforms have recently become commercially available, including augmented reality systems like Google Glass and in-home platforms like Xbox Kinect. Beyond built-in functionality, these platforms generally allow users to install and/or run applications of their choice.

While applications on these platforms provide great benefits in the common case — including social networking, photo editing and sharing, GPS-based maps and directions, and health sensing — their presence also raises security and privacy risks. For example, smartphone malware has been on the rise [155], often appearing as compromised versions of legitimate applications [13]. These trojans may request more permissions than the original applications, but users have been trained to click through permission-granting dialogs rather than choosing not to install an application that they wish to use [118]; users are further desensitized by the fact that applications often request more permissions than needed [15, 49]. In other cases, violations of user privacy occur entirely without the user's knowledge or consent: mobile and web applications routinely leak or misuse users' private data [30, 42, 78], often with the goal of tracking users for the purposes of targeted advertising and similar.

The work presented in this dissertation characterizes and tackles such security and privacy issues posed by untrusted applications on key client platforms: web browsers, modern operating systems (e.g., smartphones), and emerging continuous sensing platforms (e.g., augmented reality systems). These three platforms encompass a large fraction of the activities carried out by today's computer users and represent different points in the technology landscape, allowing us to explore proposed solutions across their similarities and differences. Taken together, the following chapters consider two significant security and privacy challenges that arise in each of these platforms:

1. *Embedded Third-Party Principals.* First, modern web and smartphone applications generally contain more than meets the eye: in addition to its own (first-party) content, an application often embeds visible and invisible third-party content. For example, websites embed script or iframes providing advertisements, social media widgets (e.g., Facebook's "Like" or Twitter's "tweet" button), Google search results, or maps; smartphone applications include third-party libraries that display advertisements or provide billing functionality. Unfortunately, embedded third-party content comes with both privacy and security implications, as discussed further below. This dissertation characterizes, evaluates, and addresses many of these concerns in the context of both web (Chapter 2) and smartphone (Chapter 4) applications.
2. *Application Permission Granting.* Second, modern client platforms, such as smartphones or emerging wearable systems (like Google Glass), commonly allow users to install and/or run potentially untrusted applications. As described in more detail below, these platforms do not allow applications to access sensitive resources (such as the complete file system or devices like the camera, microphone, or other sensors) by default. Rather, they allow applications to request, and rely on users to grant, the necessary permissions. Unfortunately, today's permission granting models provide inadequate security and usability. This dissertation develops two new models for permission granting that improve on existing approaches: user-driven access control for today's smartphone applications (Chapter 3) and world-driven access control for emerging augmented reality and other continuous sensing applications (Chapter 5).

Together, these two challenges account for many of the risks posed by untrusted applications in modern and emerging client platforms. Though each chapter of this dissertation concerns itself primarily with one of these challenges, both themes will recur throughout and build on each other. The remainder of this chapter describes and provides background for each challenge, before summarizing the contributions presented in this dissertation. By tackling these challenges, this dissertation improves the state of the art and provides a foundation for stronger security and privacy from untrusted client-side applications.

1.1 First Challenge: Embedded Third-Party Principals

Modern applications consist not only of their own content, but frequently also include visible and invisible third-party content. For example, web and smartphone applications commonly include third-party advertisements, analytics services (e.g., Google Analytics), and social sharing widgets (e.g., Facebook's "Like" button or comment feed, Google's "+1" button, or a Twitter feed). Other examples of embeddable content include search boxes and maps.

Today's platforms support the embedding of third-party principals in two primary ways. First, an application can include a third-party library that runs in the execution context of the application itself. On the web, this is done using a `<script>` HTML tag to reference a JavaScript library on another domain; this method is commonly used for web analytics platforms, such as Google Analytics (whose script is hosted at `http://google-analytics.com`). Similarly, smartphone applications can compile in third-party libraries, such as advertising libraries, that then run as part of the application itself.

The web also provides a second, more isolated option for embedding content: a website may delegate a (possibly invisible) portion of its user interface to a third party, such as an advertisement, embedded in an iframe. The browser isolates iframes from the parent webpage according to the same-origin policy [169], preventing the parent page and the iframe from interacting with each other except through explicit interfaces. Such an isolated option is not available on today's smartphone platforms — there, third-party advertisements, for example, are drawn by an included advertising library and not isolated in any way from the embedding application.

Third-party embedding has security and privacy implications both for the user of an application as well as for the application itself. First, it poses a *risk to users*. Though embedded third parties are often invisible or indistinguishable to users, their presence has privacy implications. On the web, these third parties are able to identify and collect information about users, generally in the form of some subset of web browsing history, a practice referred to as web tracking. On smartphones, third-party libraries commonly leak device identification and location information to advertising and analytics servers [67].

Second, third-party embedding poses a *risk to the embedding application*, as well as a *risk* *to the embedded content.* For third-party content running in the parent application's own execution context, the lack of isolation puts the parent application completely at risk — the embedded principal can exfiltrate sensitive data and abuse the parent application's permissions. More generally, the lack of security considerations in the user interface toolkits of many of today's platforms create cross-principal risks. For example, a malicious embedded principal (such as an advertisement) may attempt to take over a larger display area than expected. On the other hand, a malicious host may attempt to eavesdrop on input intended for embedded content or forge a user's intent to interact with it, either by tricking the user (e.g., by clickjacking) or by programmatically issuing input events. A common attack of this type is the "likejacking" attack [154], in which a host website tricks the user into clicking on a Facebook "Like" button, thereby inadvertently linking that website to his or her Facebook profile.

This dissertation first considers embedded third-party principals from the browser's perspective in Chapter 2, which characterizes, quantifies, and proposes novel defenses for web tracking. Chapter 4 then considers embedded third-party user interface content more generally and describes a modified version of Android (a smartphone operating system) that securely supports third-party content embedding.

1.2 *Second Challenge: Application Permission Granting*

Traditional desktop operating systems treat users as distinct security principals and grant applications nearly unrestricted resource access by virtue of installation. While applications may be restricted to user account privileges — requiring a user action (e.g., the use of `sudo`, entering a password, and/or agreeing to a privilege elevation prompt) to receive administrative privileges — they can freely access global resources like the file system, the clipboard, and devices like the camera without explicit permission from the user. The assumption is that by installing an application, users make the decision to trust it with access to these resources.

The emergence of the web required a new model, as users do not install web applications, but rather browse to pages, which may arbitrarily redirect the user's browser or update dynamically without the user's knowledge or explicit consent. To protect the user's machine and data in other web applications, browsers run each page in an isolated environment with limited privileges. Content from different domains is isolated according to the same-origin policy [169], and web applications cannot access local system resources (e.g., the clipboard, the file system, or the camera) without explicit user permission.

More recently, smartphone operating systems like iOS, Android, and Windows Phone have emerged and adopted a similar model, isolating multiple applications from each other and from the system itself, and granting access to sensitive resources only via explicit user permission. Modern desktop operating systems (e.g., Windows 8) have also moved in this direction, recognizing that the user's choice to install an application is not sufficient to grant that application unrestricted resource access.

From a security perspective, the restrictions placed by modern platforms on applications are an improvement over traditional desktop systems. Malicious applications can no longer freely access system resources and user data. However, applications must sometimes legitimately interact with each other and with system resources in order to provide their intended and desired functionality. To enable these interactions, systems allow applications to request permission to pierce their isolation containers.

In doing so, today's systems employ permission granting models that still provide inadequate security. In particular, existing access control and permission-granting mechanisms tend to be coarse-grained, abrasive, or inadequate. For instance, they require users to make out-of-context, uninformed decisions at install time via manifests, or they unintelligently prompt users to determine their intent. Because these permission-granting mechanisms are not usable and not well-understood by users (e.g., [50, 118]), they do not adequately protect users from malicious applications (e.g., [13, 161]).

Thus, a pressing open problem in these modern client platforms is how to allow users to grant applications permissions to sensitive resources, e.g., to privacy- and cost-sensitive devices like the camera or to user data residing in other applications. A key challenge, addressed in this dissertation, is to enable such access in a way that is non-disruptive to users while still maintaining least-privilege restrictions on applications.

This dissertation describes two novel permission granting models that are usable, least-privilege, and generalizable across sensitive resources and across platforms. In these models, the trusted platform extracts access control decisions from the user's natural interaction with the system and its applications (in *user-driven access control*, Chapter 3) or from the real-world context surrounding the user and her device (in *world-driven access control*, Chapter 5). Our techniques allow the system to make access control decisions aligned with the user's intent and expectations, without being overly permissive, and without continuously asking the user to make explicit, low-level access control decisions.

1.3 Contributions

This dissertation makes the following contributions with respect to these two challenges — embedded third-party principals and application permission granting — spanning a variety of modern and emerging client platforms:

Detecting and defending against third-party tracking on the web. Chapter 2 explores how embedded third-party principals on websites enable third-party web tracking, the practice by which websites re-identify users and collect information about their browsing history. This dissertation develops a client-side method for detecting and classifying five kinds of third-party trackers based on how they manipulate browser state, distinguishing tracking behaviors not differentiated in prior work. We use this client-side tracking detection method to perform the most comprehensive measurement study of web tracking in the wild to date. Among other contributions, our measurements reveal the prevalence of "personal trackers," embedded third-party social media widgets (like the Facebook "Like" and Twitter "tweet" buttons) that enable tracking by the associated social media site. This finding informs our design of *ShareMeNot*, a novel defense for personal trackers. Finally, we make our client-side tracking detection techniques available to users and other developers in the form of *TrackingObserver*, our tracking detection, blocking, and measurement platform.

Rethinking permission granting in modern operating systems with *user-driven access control.* Chapter 3 develops user-driven access control as a new model for permission granting in modern operating systems (e.g., smartphones). Rather than asking the user to agree to an install-time permission manifest or respond to time-of-use permission prompts, user-driven access control builds permission granting into existing user actions in the context of an application. By introducing special trusted user interface elements called access control gadgets (ACGs), we allow the operating system to extract a user's permission granting intent from the way he or she naturally interacts with any application. For example, to access the camera, an application embeds a system-provided camera button (camera ACG); when the system detects that the user has clicked on this button, it grants temporary camera access to the application. User-driven access control improves security and better matches user expectations about permission granting than existing approaches.

Securing embedded user interfaces in modern operating systems. Securely enabling access control gadgets requires considering the security of embedded third-party content — to prevent, for example, a malicious host application from programmatically clicking on a camera ACG. Chapter 4 steps back and considers more broadly the security and privacy issues that arise with embedded third-party user interfaces. Specifically, this dissertation explores the requirements for a system to support secure embedded user interfaces (UIs) by systematically analyzing existing systems like browsers, smartphones, and research systems. We develop a set of user interface toolkit techniques for supporting secure UI embedding and concretely implement and refine these techniques in *LayerCake*, a modified version of a smartphone operating system (Android). These techniques enable functionality that is not possible to implement securely on stock Android, including advertisement libraries, Facebook social plugins (e.g., the "Like" button), and access control gadgets.

Permission granting for continuous sensing platforms with *world-driven access control.* Chapter 5 considers application permission granting in the context of emerging continuous sensing platforms like augmented reality systems, in which applications rely on long-term access to sensor data and explicit user input is sparse. This dissertation introduces world-driven access control, which allows real-world objects to explicitly specify access policies; a trusted policy module on the platform senses policies in the world and automatically modifies each application's "view" accordingly (for example, blocking video frames in a sensitive location like a locker room). To convey and authenticate policies, we introduce passports, a new kind of certificate that includes both a policy and optionally the code for recognizing a real-world object. The world-driven access control approach relieves the user's permission management burden while mediating access at the granularity of objects rather than complete sensor streams.

Taken together, Chapters 2–5 provide a foundation for addressing two fundamental challenges in security and privacy from untrusted applications in modern and emerging client platforms. Chapter 6 summarizes these contributions and concludes.

Chapter 2

DETECTING AND DEFENDING AGAINST THIRD-PARTY TRACKING ON THE WEB

This chapter considers and tackles the first key challenge identified in Chapter 1, embedded third-party principals, in the context of the web. Specifically, we consider the privacy implications of embedded third-party content on websites, which enables the third party to re-identify users and collect information about their browsing history, a practice known as web tracking. While web tracking has garnered much attention in recent years, its working remained poorly understood before the work presented in this chapter, much of which was originally described in a 2012 paper [139]. This chapter characterizes and measures how mainstream web tracking occurs in the wild, and then uses these findings to motivate two tools: ShareMeNot, a new defense to prevent tracking by social media sites via widgets while still allowing those widgets to achieve their utility goals, and TrackingObserver, a web tracking detection, measurement, and blocking platform.

2.1 Motivation and Overview

Web tracking, the practice by which websites identify, and collect information about users — generally in the form of some subset of web browsing history — has become a topic of increased public debate. Before the work described in this chapter, however, the research community's knowledge of web tracking was piecemeal. There are many specific ways that identifying information might be gleaned (e.g., browser fingerprinting [39], ETags [12], and Flash cookies [153]) but little assessment of how tracking is integrated with web browsing in practice. Further complicating the situation is that the capabilities of different trackers depend strongly on their implementation. For instance, it is common for trackers like Google Analytics and Doubleclick to be mentioned in the same context (e.g., [87]) even though the former is implemented so that it cannot use unique identifiers to track users across sites while the latter can.

As the tracking arms race continues, the design of future web systems must be informed by an understanding of how web trackers retask browser mechanisms for tracking purposes. Our goal is thus to provide a comprehensive assessment of third-party tracking on the web today, where a third-party tracker is defined as a website (like `doubleclick.net`) that has its tracking code included or embedded in another site (like `cnn.com`). We focus on third-party tracking because of its potential concern to users, who may be surprised that a party with which they may or may not have chosen to interact is recording their online behavior in unexpected ways. We also explicitly focus on mainstream web tracking that uses cookies and other conventional local storage mechanisms (HTML5, Flash cookies) to compile records of users and user behavior. This is the most prevalent form of tracking today. More esoteric forms of tracking do exist — such as tracking with Etags [12], visited link coloring [77], and via the cache — and would threaten privacy if widely deployed, but they are not commonly used today. We similarly exclude inference-based browser and machine fingerprinting techniques [3, 39, 122, 179], commonly used for online fraud detection [164, 168], in favor of explicit tracking that pinpoints users with browser state.

Our approach is to detect tracking as it is observed by clients; we achieve this goal by integrating web tracking detection directly into the browser. We began by looking at how real tracker code interacts with browsers, and from there distill five distinct behavior types. In our system, we are able to distinguish these different sets of behaviors, for example classifying Google Analytics and Doubleclick as distinct.

We then developed a Firefox browser extension, TrackingTracker, to measure the prevalence of different web trackers and tracking behaviors. We later refined and extended TrackingTracker into a browser-based tracking detection platform, TrackingObserver, which we released publicly. We aimed our tool at the 500 most popular and 500 less popular websites according to the Alexa rankings, as well as real user workloads as approximated with web traces generated from publicly available AOL search logs. Our measurements reveal extensive tracking. Pages are commonly watched by more than one of the over 500 unique trackers we found. These trackers exhibit a variety of nondeterministic behaviors, including hierarchies in which one tracker hands off to another. Several trackers have sufficient penetration that they may capture a large fraction of a user's browsing activity.

Our method also allowed us to assess how today's defenses reduce web tracking. We found that popup blocking, third-party cookie blocking and the Do Not Track header thwarted a large portion of cookie-based tracking without impacting functionality in most browsers, with the exception of tracking by social media widgets. Disabling JavaScript is more effective but can significantly impact the browsing experience. Tracking by social media widgets (e.g., Facebook's "Like" button) has rapidly grown in coverage and highlights how unanticipated combinations of browser mechanisms can have unexpected effects. Informed by our understanding of this kind of tracking, as well as the inadequacy of existing solutions, we developed and released the ShareMeNot browser extension to successfully defend against it while still allowing users to interact with the widgets.

To summarize, we make several contributions. Our classification of tracking behaviors is new, and goes beyond simple notions of first- and third-party tracking. Our measurements of deployed web trackers and how much they track users give the most detailed account of which we are aware to date of tracking in the wild, as well as an assessment of the efficacy of common defenses. Our ShareMeNot browser extension provides a new defense against a practical threat. Finally, we extend and release our measurement tool in the form of TrackingObserver, a browser-based web tracking detection, measurement, and blocking platform. We now turn to additional background information in Section 2.2.

2.2 Background

Third-party web tracking refers to the practice by which an entity (the tracker), other than the website directly visited by the user (the site), tracks or assists in tracking the user's visit to the site. For instance, if a user visits `cnn.com`, a third-party tracker like `doubleclick.net` — embedded by `cnn.com` to provide, for example, targeted advertising — can log the user's visit to `cnn.com`. For most types of third-party tracking, the tracker will be able to link the user's visit to `cnn.com` with the user's visit to other sites on which the tracker is also embedded. We refer to the resulting set of sites as the tracker's *browsing profile* for that user. Before diving into the mechanisms of third-party tracking, we briefly review necessary web-related background.

2.2.1 Web-Related Background

Page fetching. When a page is fetched by the browser, an HTTP request is made to the site for a URL in a new top-level execution context for that site (that corresponds to a user-visible window with a site title). The HTTP response contains resources of several kinds (HTML, scripts, images, stylesheets, and others) that are processed for display and which may trigger HTTP requests for additional resources. This process continues recursively until loading is complete.

Execution context. A website can embed content from another domain in two ways. The first is the inclusion of an iframe, which delegates a portion of the screen to the domain from which the iframe is sourced — this is considered the *third-party domain*. The *same-origin policy* ensures that content from the two domains is isolated: any scripts running in the iframe run in the context of the third-party domain. By contrast, when a page includes a script from another domain (using `<script src=...>`), that script runs in the domain of the embedding page (the *first-party domain*), not in that of the script's source.

Client-side storage. Web tracking relies fundamentally on a website's ability to store state on the user's machine — as do most functions of today's web. Client-side state may take many forms, most commonly traditional browser cookies. A *cookie* is a triple (domain, key, value) that is stored in the browser across page visits, where domain is a web site, and key and value are opaque identifiers. Cookies that are set by the domain that the user visits directly (the domain displayed in the browser's address bar) are known as *first-party cookies*; cookies that are set by some other domain embedded in the top-level page are *third-party cookies*.

Cookies are set either by scripts running in the page using an API call, or by HTTP responses that include a Set-Cookie header. The browser automatically attaches cookies for a domain to outgoing HTTP requests to that domain, using Cookie headers. Cookies may also be retrieved using an API call by scripts running in the page and then sent via any channel, such as part of an HTTP request (e.g., as part of the URL). The same-origin policy ensures that cookies (and other client-side state) set by one domain cannot be directly accessed by another domain.

Users may choose to block cookies via their browser's settings menu. Blocking all cookies is uncommon,[1] as it makes today's web almost unusable (e.g., the user cannot log into any account), but blocking third-party cookies is commonly recommended as a first line of defense against third-party tracking.

In addition to traditional cookies, HTML5 introduced new client-side storage mechanisms for browsers. In particular, *LocalStorage* provides a persistent storage area that sites can access with API calls, isolated by the same-origin policy. Plugins like Flash are another mechanism by which websites can store data on the user's machine. In the case of Flash, websites can set *Local Storage Objects* (LSOs, also referred to as "Flash cookies") on the user's file system.

2.2.2 Background on Tracking

Web tracking is highly prevalent on the web today. From the perspective of website owners and of trackers, it provides desirable functionality, including personalization, site analytics, and targeted advertising. A recent study [55] claims that the negative economic impact of preventing targeted advertising — or the underlying tracking mechanisms that enable it — is significant. From the perspective of a tracker, the larger a browsing profile it can gather about a user, the better service it can provide to its customers (the embedding websites) and to the user herself (e.g., in the form of personalization).

From the perspective of users, however, larger browsing profiles spell greater loss of privacy. A user may not, for instance, wish to link the articles he or she views on a news site with the type of adult sites he or she visits, much less reveal this information to an unknown third party. Even if the user is not worried about the particular third party, this data may later be revealed to unanticipated parties through court orders or subpoenas.

Despite the prevalence of tracking and the resulting public and media outcry — primarily in the United States and in Europe — there is a lack of clarity about how tracking works,

[1]On May 22, 2014, the Gibson Research Corporation cookie statistics page (`http://www.grc.com/cookies/stats.htm`) showed that almost 100% of 52,029 unique visitors in the previous week had first-party cookies enabled. Earlier measurements, during the original execution of this work in 2011, showed similar results.

how widespread the practice is, and the scope of the browsing profiles that trackers can collect about users. Tracking is often invisible; tools like the Ghostery Firefox add-on[2] aim to provide users with insight into the trackers they encounter on the web. What these tools do not consider, however, are the differences between types of trackers, their capabilities, and the resulting scope of the browsing profiles they can compile. For example, Google Analytics is commonly considered to be one of the most prominent trackers. However, it does not have the ability to create cross-site browsing profiles using the unique identifiers in its cookies. Thus, its prevalence is not correlated with the size of the browsing profiles it creates.

Storage and communication. Our study focuses on *explicit* tracking mechanisms — tracking mechanisms that use assigned, unique identifiers per user — rather than *inferred* tracking based on browser and machine fingerprinting. Other work (e.g., [122, 179]) has studied the use of fingerprinting to pinpoint a host with high accuracy. More specifically, all trackers we consider have two key capabilities:

1. The ability to store a pseudonym (unique identifier) on the user's machine.
2. The ability to communicate that pseudonym, as well as visited sites, back to the tracker's domain.

The pseudonym may be stored using any of the client-side storage mechanisms described in Section 2.2.1 — in a conventional browser cookie, in HTML5 LocalStorage, and in Flash LSOs, as well as in more exotic locations such as in ETags. There are multiple ways in which the browser may communicate information about the visited site to the tracker, e.g., implicitly via the *HTTP Referrer header* or explicitly via tracker-provided JavaScript code that directly transmits the results of an `document.referrer` API call. In some cases, a script running within a page might even communicate the visited page information in the GET or POST parameters of a request to a tracker's domain. For example, a tracker embedded on a site might access its own cookie and the referring page, and then pass this information on to another tracker with a URL of the form `http://tracker2.com/track?cookie_value=123&site=site.com`.

[2]`http://www.ghostery.com`

Different scales of tracking. Depending on the behaviors exhibited and mechanisms used by a tracker, the browsing profiles it compiles can be *within-site* or *cross-site*. Within-site browsing profiles link the user's browsing activity on one site with his or her other activity only on that site, including repeat visits and how the website is traversed, but not to visits to any other site. Cross-site browsing profiles link visits to multiple different websites to a given user (identified by a unique identifier or linked by another technique [85, 179]).

Behavioral methodology. In this paper, we consider tracking behavior that is observable from the client, that is, from the user's browser. Thus, we do not distinguish between "can track" and "does track" — that is, we analyze trackers according to the capabilities granted by the behaviors we observe and not, for example, the privacy policies of the tracking sites.

From the background that we have introduced in this section, we step back and consider, via archetypical examples, the set of properties exhibited by trackers (Section 2.3.1); from these properties we formulate a classification of tracking behavior in Section 2.3.2.

2.3 Classifying Web Tracking Behavior

All web trackers that use unique identifiers are often bundled into the same category. However, in actuality diverse mechanisms are used by trackers, resulting in fundamentally different tracking capabilities. Our observations, based both on manual investigations and automated measurements, lead us to believe that it is incorrect to bundle together different classes of trackers — for example, Google Analytics is a within-site tracker, while Doubleclick is a cross-site tracker. To rigorously evaluate the tracking ecosystem, we need a framework for differentiating different tracker types. We develop such a framework here (Section 2.3.2). To inform this framework, we first dive deeply into an investigatory analysis of how tracking occurs today (Section 2.3.1), where we identify different properties of different trackers. We use our resulting framework as the basis for our measurements in Section 2.4.

2.3.1 Investigating Tracking Properties

In order to understand patterns of tracking behavior, we must first understand the properties of different trackers. We present several archetypal tracking examples here and, from each,

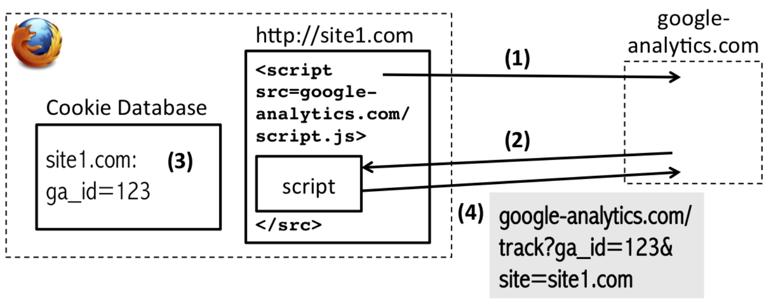


Figure 2.1: **Case Study: Third-Party Analytics.** Websites commonly use third-party analytics engines like Google Analytics (GA) to track visitors. This process involves (1) the website embedding the GA script, which, after (2) loading in the user's browser, (3) sets a site-owned cookie. This cookie is (4) communicated back to GA along with other tracking information.

extract a set of core properties.

Throughout this discussion, we will refer to cookies set under a tracker's domain as *tracker-owned* cookies. We introduce this term rather than using "third-party cookies" because a given cookie can be considered a first-party or a third-party cookie depending on the current browsing context. (For example, Facebook's cookie is a first-party cookie when the user visits `facebook.com`, but it is a third-party cookie when a Facebook "Like" button is embedded on another webpage.) Similarly, a cookie set under the domain of the website embedding a tracker is a *site-owned* cookie.

Third-Party Analytics

For websites that wish to analyze traffic, it has become common to use a third-party analytics engine such as Google Analytics (GA) in lieu of collecting the data and performing the analysis themselves. The webpage directly visited by the user includes a library (in the form of a script) provided by the analytics engine on pages on which it wishes to track users (see Figure 2.1).

To track repeat visitors, the GA script sets a cookie on the user's browser that contains a unique identifier. Since the script runs in the page's own context, the resulting cookie is site-owned, not tracker-owned. The GA script transfers this identifier to `google-analytics.com` by making explicit requests that include custom parameters in the URL containing information like the embedding site, the user identifier (from the cookie), and system information (operating system, browser, screen resolution, geographic information, etc.).

Because the identifying cookie is site-owned, identifiers set by Google Analytics across different sites are different. Thus, the user will be associated with a different pseudonym on the two sites, limiting Google Analytics's ability to create a cross-site browsing profile for that user.

Tracker properties. We extract the following set of properties defining trackers like Google Analytics:

1. The tracker's script, running in the context of the site, sets a site-owned cookie.
2. The tracker's script explicitly leaks the site-owned cookie in the parameters of a request to the tracker's domain, circumventing the same-origin policy.

Third-Party Advertising

The type of tracking most commonly understood under "third-party tracking" is tracking for the purpose of targeted advertising. As an example of this type of tracking scenario, we consider Google's advertising network, Doubleclick. Figure 2.2 shows an overview of this scenario.

When a page like `site1.com` is rendered on the user's browser, Doubleclick's code will choose an ad to display on the page, e.g., as an image or as an iframe. This ad is hosted by `doubleclick.net`, not by the embedding page (`site1.com`). Thus, the cookie that is set as the result of this interaction (again containing a unique identifier for the user) is tracker-owned. As a result, the same unique identifier is associated with the user whenever any site embeds a Doubleclick ad, allowing Doubleclick to create a cross-site browsing profile for that user.

Tracker properties. We extract the following properties defining trackers like Doubleclick:

1. The tracker sets a tracker-owned cookie, which is then automatically included with any requests to the tracker's domain.

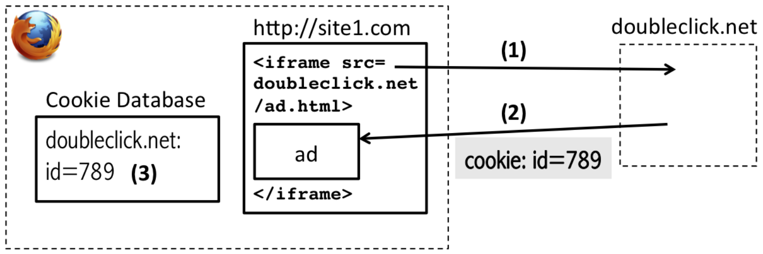


Figure 2.2: **Case Study: Third-Party Advertising.** When a website (1) includes a third-party ad from an entity like Doubleclick, Doubleclick (2-3) sets a tracker-owned cookie on the user's browser. Subsequent requests to Doubleclick from any website will include that cookie, allowing it to track the user across those sites.

2. The tracker-owned cookie is set by the tracker in a third-party position — that is, the user never visits the tracker's domain directly.

Third-Party Advertising with Popups

A commonly recommended first line of defense against third-party tracking like that done by Doubleclick is third-party cookie blocking. However, in most browsers, third-party cookie blocking applies only to the setting, not to the sending, of cookies (in Firefox, it applies to both). Thus, if a tracker is able to maneuver itself into a position from which it can set a first-party cookie, it can avoid the third-party cookie blocking defense entirely.

We observed this behavior from a number of trackers, such as `insightexpressai.com`, which opens a popup window when users visit `weather.com`. While popup windows have other benefits for advertising (e.g., better capturing a user's attention), they also put the tracker into a first-party position without the user's consent. From there, the tracker sets and reads first-party cookies, remaining unaffected by third-party cookie blocking.

Tracker properties. We extract the following properties defining trackers like Insight Express:

1. The tracker forces the user to visit its domain directly, e.g., with a popup or a redirect, allowing it to set its tracker-owned cookie from a first-party position.
2. The tracker sets a tracker-owned cookie, which is then automatically included with

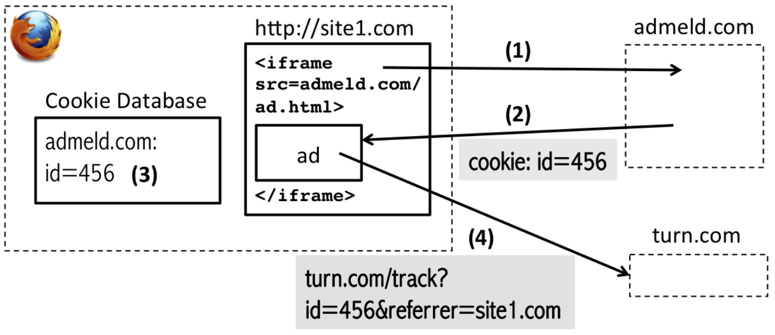


Figure 2.3: **Case Study: Advertising Networks.** As in the ordinary third-party advertising case, a website (1-2) embeds an ad from Admeld, which (3) sets a tracker-owned cookie. Admeld then (4) makes a request to another third-party advertiser, Turn, and passes its own tracker-owned cookie value and other tracking information to it. This allows Turn to track the user across sites on which Admeld makes this request, without needed to set its own tracker-owned state.

any requests to the tracker's domain when allowed by the browser.

Third-Party Advertising Networks

While, from our perspective, we have limited insights into the business models of third-party advertisers and other trackers, we can observe the effects of complex business relationships in the requests to third-parties made by the browser. In particular, trackers often cooperate, and it is insufficient to simply consider trackers in isolation.

As depicted in Figure 2.3, a website may embed one third-party tracker, which in turn serves as an aggregator for a number of other third-party trackers. We observed this behavior to be common among advertising networks. For example, `admeld.com` is often embedded by websites, and it makes further requests to trackers like `turn.com` and `invitemedia.com`. In these requests, `admeld.com` includes the information necessary to track the user, including the top-level page and the pseudonym from `admeld.com`'s own tracker-owned cookie. This means that `turn.com` does not need to set its own client-side state, but rather can rely entirely on `admeld.com`.

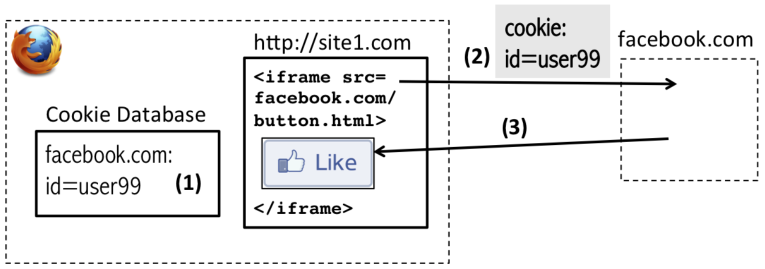


Figure 2.4: **Case Study: Social Widgets.** Social sites like Facebook, which users visit directly in other circumstances — allowing them to (1) set a cookie identifying the user — expose social widgets such as the "Like" button. When another website embeds such a button, the request to Facebook to render the button (2-3) includes Facebook's tracker-owned cookie. This allows Facebook to track the user across any site that embeds such a button.

Tracker properties. We extract the following properties defining trackers of this type:

1. The tracker is not embedded by the first-party website directly, but referred to by another tracker on that site.
2. The tracker relies on information passed to it in a request by the cooperating tracker.

Third-Party Social Widgets

An additional class of trackers doubles as sites that users otherwise visit intentionally, and often have an account with. Many of these sites, primarily social networking sites, expose social widgets like the Facebook "Like" button, the Twitter "tweet" button, the Google "+1" button and others. These widgets can be included by websites to allow users logged in to these social networking sites to like, tweet, or +1 the embedding webpage.

Figure 2.4 overviews the interaction between Facebook, a site embedding a "Like" button, and the user's browser. The requests made to `facebook.com` to render this button allow Facebook to track the user across sites just as Doubleclick can — though note that unlike Doubleclick, Facebook sets its tracker-owned cookie from a first-party position when the user voluntarily visits `facebook.com`.

Category	Name	Profile Scope	Summary	Example	Visit Directly?
A	Analytics	Within-Site	Serves as third-party analytics engine for sites.	Google Analytics	No
B	Vanilla	Cross-Site	Uses third-party storage to track users across sites.	Doubleclick	No
C	Forced	Cross-Site	Forces user to visit directly (e.g., via popup or redirect).	InsightExpress	Yes (forced)
D	Referred	Cross-Site	Relies on a B, C, or E tracker to leak unique identifiers.	Invite Media	No
E	Personal	Cross-Site	Visited directly by the user in other contexts.	Facebook	Yes

Table 2.1: **Classification of Tracking Behavior.** Trackers may exhibit multiple behaviors at once, with the exception of Behaviors B and E, which depend fundamentally on a user's browsing behavior: either the user visits the tracker's site directly or not.

Tracker properties. We extract the following set of properties, where the important distinction to the Doubleclick scenario is the second property:

1. The tracker makes use of a tracker-owned cookies.
2. The user voluntarily visits the tracker's domain directly, allowing it to set the tracker-owned cookie from a first-party position.

2.3.2 A Classification Framework

We now present a classification framework for web trackers based on observable behaviors. This is in contrast to past work that considered business relationships between trackers and the embedding website rather than observable behaviors [74] and past work that categorized trackers based on prevalence rather than user browsing profile size, thereby commingling within-site and cross-site tracking [87]. In particular, from our manual investigations we distilled five tracking behavior types; we summarize these behaviors below and in Table 2.1. Table 2.2 captures the key properties from Section 2.3.1 and their relationships to these behavioral categories. In order to fall into a particular behavior category, the tracker *must* exhibit (at least) all of the properties indicated for that category in Table 2.2. A single

	Behavior				
Property	*A*	*B*	*C*	*D*	*E*
Tracker sets site-owned (first-party) state.	✓				
Request to tracker leaks site-owned state.	✓				
Third-party request to tracker includes tracker-owned state.		✓	✓		✓
Tracker sets its state from third-party position; user never directly visits tracker.		✓			
Tracker forces user to visit it directly.			✓		
Relies on request from another B, C, or E tracker (not from the site itself).				✓	
User voluntarily visits tracker directly.					✓

Table 2.2: **Tracking Behavior by Mechanism.** In order for a tracker to be classified as having a particular behavior (A, B, C, D, or E), it *must* display the indicated property. Note that a particular tracker may exhibit more than one of these behaviors at once.

tracker may exhibit more than one of these behaviors, as we discuss in more detail below.

1. **Behavior A (Analytics)**: The tracker serves as a third-party analytics engine for sites. It can only track users within sites.
2. **Behavior B (Vanilla)**: The tracker uses third-party storage that it can get and set only from a third-party position to track users across sites.
3. **Behavior C (Forced)**: The cross-site tracker forces users to visit its domain directly (e.g., popup, redirect), placing it in a first-party position.
4. **Behavior D (Referred)**: The tracker relies on a B, C, or E tracker to leak unique identifiers to it, rather than on its own client-side state, to track users across sites.
5. **Behavior E (Personal)**: The cross-site tracker is visited by the user directly in other contexts.

This classification is based entirely on tracker behavior that can be observed from the client side. Thus, it does not capture backend tracking behavior, such as correlating a user's browsing behavior using browser and machine fingerprinting techniques, or the backend exchange of data among trackers. Similarly, the effective type of a tracker encountered by a user depends on the user's own browsing behavior. In particular, the distinction between Behavior B and Behavior E depends on whether or not the user ever directly visits the tracker's domain.

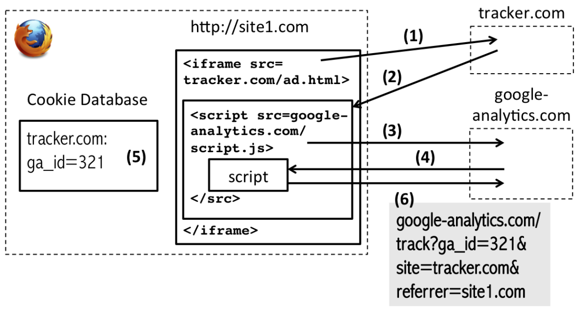

http://site1.com
Cookie Database
tracker.com: ga_id=321
(5)
<iframe src= tracker.com/ad.html>
<script src=google-analytics.com/ script.js>
script
</src>
</iframe>
tracker.com
google-analytics.com
(1)
(2)
(3)
(4)
(6)
google-analytics.com/ track?ga_id=321& site=tracker.com& referrer=site1.com

Figure 2.5: **Combining Behavior A and Behavior D.** When a Behavior A tracker like Google Analytics is embedded by another third-party tracker, rather than by the visited website itself, Behavior D emerges. The site-owned cookie that GA sets on `tracker.com` becomes a tracker-owned cookie when `tracker.com` is embedded on `site1.com`. The tracker then passes this identifier to Google Analytics, which gains the ability to track the user across all sites on which `tracker.com` is embedded.

Combining behaviors. Most of these behaviors are not mutually exclusive, with the exception of Behavior B (Vanilla) and Behavior E (Personal) — either the user directly visits the tracker's domain at some point or not. That is, a given tracker can exhibit different behaviors on different sites or multiple behaviors on the same site. For example, a number of trackers — such as `quantserve.com` — act as both Behavior A (Analytics) and Behavior B (Vanilla) trackers. Thus, they provide site analytics to the embedding sites in addition to gathering cross-site browsing profiles (for the purposes of targeted advertising or additional analytics).

Through our analysis, we identified what was to us a surprising combination of behaviors — Behavior A (Analytics) and Behavior D (Referred) — by which a within-site tracker unintentionally gains cross-site tracking capabilities. We discovered this combination during our measurement study. For example, recall that Google Analytics is a within-site, not a cross-site, tracker (Behavior A). However, suppose that `tracker.com` uses Google Analytics for its own on-site analytics, thus receiving a site-owned cookie with a unique identifier. If `tracker.com` is further embedded on another site, this same cookie *becomes a tracker-owned cookie*, which is the same across all sites on which `tracker.com` is embedded. Now, when the usual request is made to `google-analytics.com` from `tracker.com` when it is embedded, *Google Analytics becomes a Behavior D tracker* — a cross-site tracker. Figure 2.5 shows an overview of this scenario. Note that we did not observe many instances of this in practice, but it is interesting to observe that within-site trackers can become cross-site trackers when different parties interact in complex ways. This observation is further evidence of the fact that the tracking ecosystem is complicated and that it is thus difficult to create simple, sweeping technical or policy solutions.

Robustness. We stress that this classification is agnostic of the practical manifestation of the mechanisms described above — that is, client-side storage may be done via cookies or any other mechanism, and information may be communicated back to the tracker in any way. This separation of semantics from mechanism makes the classification robust in the face of the evolution of specific client-side storage techniques used by trackers.

2.4 Detecting Trackers

Based on this classification framework, we created a tool — TrackingTracker — that automatically classifies trackers according to behavior observed on the client-side. TrackingTracker runs as a Firefox add-on, interposes on all HTTP(S) requests, and examines conventional cookies, HTML5 Local Storage, and Flash LSOs to detect and categorize trackers. It has support for crawling a list of websites to an arbitrary link depth and for performing a series of search engine keyword searches and visiting the top hit of the returned search results. We used this tool to perform a series of analyses between September and October of 2011. Though we repeated many of these measurements in November 2013, we found that the tracking ecosystem had not substantially changed in the meantime. Thus, for the purposes of this dissertation, we report only our 2011 findings in detail here.

In presenting the results of these measurements, we make a distinction between *pages* and *domains*. Two pages may belong to the same domain (e.g., `www.cnn.com/article1` and `www.cnn.com/article2`). Which we use depends on whether we are interested in the characteristics of websites (domains) or in specific instances of tracking behavior (pages).

Note that the tracking behavior that we observe in our measurements is a lower bound, for several reasons. First, we do not log into any sites or click any ads or social widgets, which we have observed in small case studies to occasionally trigger additional tracking behavior. Second, we have observed that tracking behavior can be nondeterministic, largely due to the interplay of Behavior B (Vanilla) and Behavior D (Referred) trackers; we generally visit pages only twice (see below), which may not trigger all trackers embedded by a given website.

Finally, the mere presence of a cookie (or other storage item) does not by itself give a tracker the ability to create a browsing profile — the storage item must contain a unique identifier. It is difficult or impossible to identify unique identifiers with complete certainty (we do not reverse-engineer cookie strings), but we identify and remove any suspected trackers whose cookies or other storage contain identical values across multiple measurements that started with a clean browser. We also remove trackers that only use session cookies, though we note that these can equally be used for tracking as long as the browser remains open.

2.4.1 Tracking on Popular Sites

We collected a data set using the top 500 websites (international) from Alexa as published on September 19, 2011. We also visited four random links on each of the 500 sites that stayed within that site's domain. We visited and analyzed a total of 2098 unique pages for this data set; we did not visit a full 2500 unique pages because some websites do not have four within-domain links, some links are broken or redirect to other domains or to the same page, etc. This process was repeated twice: once starting with a clean browser, and once more after priming the cache and cookie database (i.e., without first clearing browser state). This experimental design aims to ensure that trackers that may only set but not read state the first time they are encountered are properly accounted for by TrackingTracker on the second run. The results we report include tracking behavior measured only on the second run.

Most of the 2098 pages (500 domains) embed trackers, often several. Indeed, the average number of trackers on the 1655 pages (457 domains) that embed at least one tracker is over

Tracker Type	Top 500 Sites #	Top 500 Sites Instances (Min, Max)	Non-Top 500 Sites #	Non-Top 500 Sites Instances (Min, Max)
A	17	49 (1, 9)	10	34 (1, 18)
A B	18	152 (1, 21)	11	104 (1, 37)
A B D	1	317 (317, 317)	1	155 (155, 155)
A E	8	47 (1, 17)	2	25 (5, 20)
A E D	1	21 (21, 21)	–	–
A D	3	902 (1, 896)	2	908 (55, 853)
B	357	3322 (1, 375)	299	3734 (1, 777)
B C	3	79 (6, 64)	–	–
B D	8	703 (1, 489)	7	60 (1, 25)
E	101	1564 (1, 397)	41	1569 (1, 446)
E C	1	34 (34, 34)	1	23 (23, 23)
E D	1	1 (1, 1)	1	417 (417, 417)
C	4	4 (1, 1)	4	4 (1, 1)
D	1	(69, 69)	3	60 (1, 57)
Total	524	7264 (1, 896)	382	7093 (1, 853)

Tracker Type	Popups Blocked #	Popups Blocked Instances (Min, Max)	Cookies Blocked #	Cookies Blocked Instances (Min, Max)	No JavaScript #	No JavaScript Instances (Min, Max)	DNT Enabled #	DNT Enabled Instances (Min, Max)
A	17	34 (1, 10)	40	158 (1, 38)	–	–	10	39 (1, 9)
A B	20	338 (1, 123)	–	–	–	–	14	105 (1, 17)
A B D	1	319 (319, 319)	–	–	–	–	1	274 (274, 274)
A E	8	51 (1, 20)	10*	95 (1, 23)	–	–	6	33 (1, 17)
A E D	1	19 (19, 19)	–	–	–	–	1	18 (18, 18)
A D	2	906 (10, 896)	1	844 (844, 844)	–	–	2	900 (81, 819)
B	336	2859 (1, 382)	1	15 (15, 15)	161	1697 (1, 263)	320	2613 (1, 305)
B C	3	29 (3, 22)	5*	48 (2, 21)	–	–	6	47 (2, 15)
B D	22	1235 (1, 494)	–	–	2	23 (10,13)	13	1299 (3, 551)
E	101	1625 (1, 405)	96*	1509 (1, 383)	49	707 (1, 195)	100	1412 (1, 338)
E C	–	–	–	–	–	–	1	31 (31, 31)
E D	1	5 (5, 5)	1*	1 (1, 1)	–	–	1	4 (4, 4)
C	–	–	5	8 (1, 4)	–	–	7	13 (1, 4)
D	2	80 (1, 79)	1*	71 (71, 71)	–	–	1	42 (42, 42)
Total	514	7505 (1, 896)	160	2749 (1, 844)	212	2427	483	6830 (1, 819)

Table 2.3: **Measurement Results.** Each set of columns reports the results for the specified measurement, each run with the Alexa top 500 sites except the Non-Top dataset. The lefthand column for each set reports the number of unique trackers of each type observed; the righthand column reports the number of occurrences of that tracker type. A value of `X (Y, Z)` in that column means that `X` occurrences of trackers of this type were observed; the minimum number of occurrences of any unique tracker was `Y` and the maximum `Z`. Values with asterisks would be zero (or shifted to another type) for Firefox users, due to that browser's stricter third-party cookie blocking policy. Some of the variation in counts across runs is due to the nondeterminism of tracking behavior.

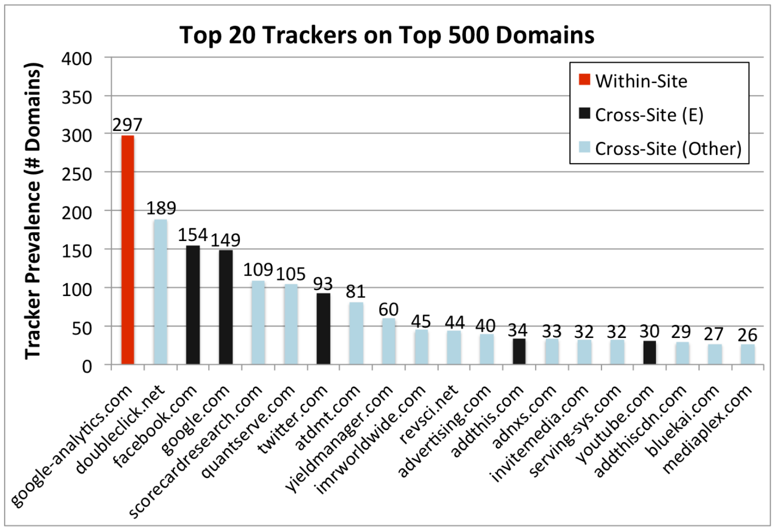


Figure 2.6: **Prevalence of Trackers on Top 500 Domains.** Trackers are counted on domains, i.e., if a particular tracker appears on two pages of a domain, it is counted once.

4.5 (over 7). Of these, 1469 pages (445 domains) include at least one cross-site tracker.

Overall, we found a total of 524 unique trackers appearing a cumulative 7264 times. Figure 2.6 shows the twenty top trackers across the 500 top domains. This graph considers websites as domains — that is, if a given tracker was encountered on two pages of a domain, it is only counted once in this graph. The most prevalent tracker is Google Analytics, appearing on almost 300 of the 500 domains — recall that it is a within-site tracker, meaning that it cannot link users' visits across these pages using cookies. The most popular cross-site tracker that users don't otherwise visit directly is Doubleclick (also owned by Google), which can track users across almost 40% of the 500 most popular sites. The most popular Behavior E tracker (domains that are themselves in the top 500) is Facebook, followed closely by Google, both of which are found on almost 30% of the top sites.

Recall that a tracker may exhibit different behavior across different occurrences, often due to varying business and embedding relationships. We thus consider occurrences in addition to unique trackers; this data is reported in the first set of columns in Table 2.3. In this table, a tracker classified as, for instance, type A B D, may exhibit different combinations of the three behaviors at different times.

We find that most trackers behave uniformly across occurrences. For example, the most common tracking behaviors are Behavior B (Vanilla) and Behavior E (Personal), which these trackers exhibit uniformly. Some trackers, however, exhibit nonuniform behavior. For instance, sites may choose whether or not to use Quantserve for on-site analytics (Behavior A) in addition to including it as a Behavior B (cross-site) tracker. Thus, Quantserve may sometimes appear as Behavior A, sometimes as Behavior B, and sometimes as both. When other trackers include Quantserve, it can also exhibit Behavior D (Referred) behavior. Similarly, Google Analytics generally exhibits Behavior A behavior, setting site-owned state on a site for which it provides third-party analytics. However, when a third-party tracker uses Google Analytics itself, as described in Section 2.3.2, Google Analytics is put into the position of a Behavior D (cross-site) tracker.

Other Storage Mechanisms

HTML5 LocalStorage. Contrary to expectations that trackers are moving away from cookies to other storage mechanisms, we found remarkably little use of HTML5 LocalStorage by trackers (though sites themselves may still use it for self-administered analytics). Of the 524 unique trackers we encountered on the Alexa top 500 sites, only eight of these trackers set any LocalStorage at all. Five contained unique identifiers, two contained timestamps, and one stored a user's unsubmitted comments in case of accidental navigation away from the page.

We observed both Behavior A and Behavior B behavior using LocalStorage. Notably, we discovered that `taboolasyndication.com` and `krxd.net` set site-owned LocalStorage instead of browser cookies for Behavior A purposes.

Of the five trackers that set unique identifiers in LocalStorage, all duplicated these values in cookies. When the same identifier is stored in multiple locations, the possibility of *respawning* is raised: if one storage location is cleared, the tracker can repopulate it with the same value from the other storage location. Respawning has been observed several times in the wild [12, 153] as a way to subvert a user's intention not to be tracked and is exemplified by the proof-of-concept evercookie.[3]

We manually checked for respawning behavior in these five cases and found that one tracker — `tanx.com` — indeed repopulated the browser cookies from LocalStorage when cleared. We also noticed that `twitter.com`, which set a uniquely identifying "guest id" on the machines of users that are not logged in, did not repopulate the cookie value — however, it did store in LocalStorage the entire history of guest ids, allowing the user's new guest id to be linked to the old one. This may not be intentional on Twitter's part, but the capability for tracking in this way — equivalent to respawning — exists.

We also observed reverse respawning, that is, repopulating LocalStorage from cookies when cleared. We observed this in three of the five cases, and note that it may not be intentional respawning but rather a function of when LocalStorage is populated (generally after checking if a cookie is set and/or setting a cookie).

Flash storage. Flash LSOs, or "cookies", on the other hand, are more commonly used to store unique tracking identifiers. Nevertheless, despite media buzz about identifier respawning from Flash cookies, we find that most unique identifiers in Flash cookies do not serve as backups to traditional cookies; only nine of the 35 trackers with unique identifiers in Flash cookies duplicate these identifiers across Flash cookies and traditional cookies.

For these nine trackers, we tested manually for respawning behavior as described above. We observed Flash-to-cookie respawning in six cases and cookie-to-Flash respawning in seven.

In one interesting case, we found that while the Flash cookie for `sodahead.com` does not appear to match the browser cookie, it is named `enc_data` and may be an encrypted version of the cookie value. Indeed, `sodahead.com` respawned the browser cookie from the Flash cookie. Furthermore, the respawned cookie was a session cookie that would ordinarily expire automatically when the browser is closed. This example demonstrates that it is not sufficient to inspect stored values but that respawning must be determined behaviorally.

[3]`http://samy.pl/evercookie/`

Cookie Leaks, Countries, and More

Throughout our study, we made a number of interesting non-quantitative observations; we describe these here.

Frequent cookie leaks. We observed a large number of cookie leaks, i.e., cookies belonging to one domain that are included in the parameters of a request to another domain, thereby circumventing the same-origin policy. Fundamentally, cookie leaking enables an additional party to gain tracking capabilities that it would not otherwise have. In addition to Behavior A leaks (the leaking of site-owned state set by the tracker's code to the tracker as a third-party) and Behavior D leaks (to enable additional trackers), we observed cookie leaks indicative of business relationships between two (or few) parties.

For example, `msn.com` and `bing.com`, both owned by Microsoft, use cookie leaking mechanisms within the browser to share cookies with each other, even when the user does not visit both sites as part of a contiguous browsing session. This enables Microsoft to track a unique user across both MSN and Bing, as well as across any site that may embed one of the two.

As another example, we noticed that when a website includes both Google AdSense (a product that allows the average website owner to embed ads without a full-fledged Doubleclick contract) as well as Google Analytics, the AdSense script makes requests to Doubleclick to fetch ads. These requests include uniquely identifying values from the site's Google Analytics cookies. This practice gives Google the capability to directly link the unique identifier used by Doubleclick to track the user across sites with the unique Google Analytics identifier used to track the user's visits to this particular site. While this does not increase the size of the browsing profile that Doubleclick can create, it allows Google to relink the two profiles if the user ever clears client-side state for one but not the other.

Origin countries. In exploring the use of LocalStorage and Flash cookies, we found that trackers from different regions appear to exhibit different behaviors. The only tracker to respawn cookies from LocalStorage comes from a Chinese domain, and of the eight trackers involved in respawning to or from Flash cookies, four are US, two are Chinese, and two are

Russian. The Chinese and Russian trackers seem to be overrepresented compared to their fraction in the complete set of observed trackers.

Tracker clustering. While many of the top trackers are found across sites of a variety of categories and origins, we observed some trackers to cluster around related sites. For example, in the Alexa top 500, `traffichaus.com` and `exoclick.com` are found only on adult sites (on five and six of about twenty, respectively). Similarly, some trackers are only found on sites of the same geographic origin — e.g., `adriver.ru` is found only on Russian sites and `wrating.com` only on Chinese sites.

Trackers interact with two types of users. We observed that Behavior B (Vanilla) and Behavior C (Forced) trackers sometimes do not set tracking cookies when their websites are visited directly — unlike Behavior E tracker like Facebook, which by definition set state when they are visited. In other words, for example, `turn.com` sets a third-party tracking cookie when it is embedded on another website, but not when the user visits `turn.com` directly. Some trackers, in fact, use different domains for their own homepages than for their tracking domains (e.g., visiting `doubleclick.net` redirects to `google.com/doubleclick`). Trackers that exhibit these behaviors can never be Behavior E trackers, even if the users directly visits their sites. We can only speculate about the reasons for these behaviors, but we observe that trackers interact with two types of users: users whom they track from a third-party position, and users who are their customers and who visit their website directly.

2.4.2 Comparison to Less Popular Sites

The measurements presented thus far give us an intuition about the prevalence and behavior of trackers on popular sites. Since it is possible that different trackers with different behavior are found on less popular sites, we collected data for non-top sites as well. In particular, we visited 500 sites from the Alexa top million sites, starting with site #501 and at intervals of 100. As in the top 500 case, we visited 4 random links on each page, resulting in a total of 1959 unique pages visited.

In this measurement, we observed 7093 instances of tracking across 382 unique trackers, summarized in the second set of columns in Table 2.3. Figure 2.7 shows the top 20 trackers

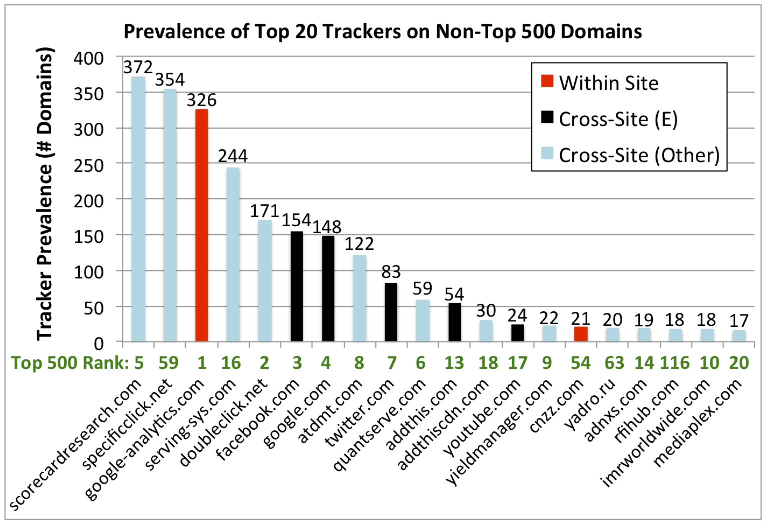


Figure 2.7: **Tracker Prevalence on Non-Top 500 Domains.** Trackers are counted on domains, i.e., if a particular tracker appears on two pages of a domain, it is counted once.

(counted by domains) for this measurement. The numbers below each bar indicate the rank for the tracker in the top 500 domains. Note that Google Analytics and Doubleclick are no longer ranked as high, but in absolute numbers appear a similar number of times. ScorecardResearch and SpecificClick appear to be highly prevalent among among these less popular sites.

Among the non-top 500 sites, we observed less LocalStorage use — only one of the eight users of LocalStorage in the top 500 sites reappeared (`disqus.com`, which stored comment drafts); we saw one additional instance of LocalStorage, `contextweb.com`, which stored a unique value but does not duplicate it in the browser cookie. We also observed fewer Flash cookies set (68 total across all sites and trackers, compared to 110 in the top 500 measurement), finding one additional tracker — `heias.com` — that respawns its browser cookie value from the Flash cookie.

2.4.3 Real Users: Using the AOL Search Logs

In order to better approximate a real user's browsing history, we collected data using the 2006 AOL search query logs [129]. We selected 35 random users (about 1%) from the

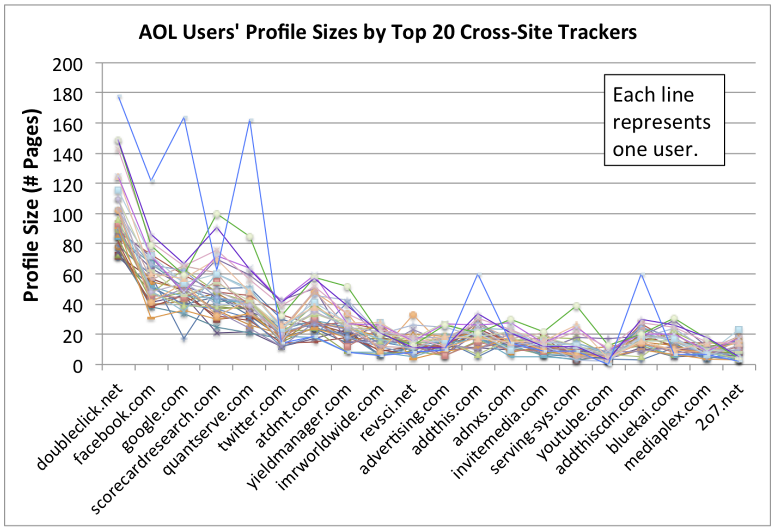


Figure 2.8: **Browsing Profiles for 35 AOL Users.** We report the measured profile size for each user for the 20 top trackers from the top 500, using 300 unique queries (an average of 253 unique pages visited) per user.

3447 users with at least 300 unique queries (not necessarily clickthroughs). For each of these randomly selected users, we submitted to a search engine the first 300 of their unique queries and visited the top search result for each. This resulted in an average of 253 unique pages per user.

For the AOL users, we are interested in the size of the browsing profiles that trackers can create. Here we must consider exactly how we define “profile”. In particular, a tracker receives information about the domains a user visits, the pages a user visits, and the individual visits a user makes (i.e., returning to the same page at a later time). A user may be concerned about the privacy of any of these sets of information; in the context of this study, we consider unique pages. That is, we consider the size of a browsing profile compiled by a given tracker to be the number of unique pages on which the user encountered that tracker. The reason for using pages instead of visits is that using search logs to approximate real browsing behavior involves making multiple visits to pages that a real user might not make — e.g., multiple unique queries may result in the same top search hit, which TrackingTracker will visit but a real user may not, depending on why a similar query was repeated. Though TrackingTracker may thus visit multiple pages more than once, giving more trackers the opportunity to load on that page, this is balanced by the fact that we, as before, visit each page once before recording measurements in order to prime the cache and the cookie database.

In order to compare AOL users, we focus on the top 20 cross-site trackers from the Alexa top 500 measurement. That is, we take all 19 cross-site trackers from Figure 2.6 as well as `serving-sys.com`, the next-highest ranked cross-site tracker. Figure 2.8 shows the size of the profile compiled by each tracker for each of the 35 users.

We find that Doubleclick can track a user across (on average) 39% of the pages he or she visited in these browsing traces — and a maximum of 66% of the pages visited by one user. The magnitude of these percentages may be cause for concern by privacy-conscious users. Facebook and Google can track users across an average of 23% and 21% of these browsing traces (45% and 61% in the maximum case), respectively. As many users have and are logged into Facebook and Google accounts, this tracking is likely not to be anonymous.

Two data points of note are the large profile sizes for `google.com` and `quantserve.com` for one of the users. These spikes occur because that user visited a large number of pages on the same domain (`city-data.com`), which embeds Google Maps and Quantserve.

From this data, we observe that, in general, the ranking of the trackers in the top 500 corresponds with how much real users may encounter them. In particular, `doubleclick.net` remains the top cross-site tracker; the prominence of `scorecardresearch.com` in the non-top 500 is not reflected here, perhaps because top search hits are likely biased towards more popular sites.

2.5 Defenses

In this section, we explore existing defenses against tracking in the context of our classification. We then present measurement results collected using the Alexa top 500 with standard defenses enabled. Finally, we propose an additional defense — implemented in the form of our browser add-on ShareMeNot — that aims to protect users from Behavior E tracking. Again, unless otherwise noted, we refer to observations in September/October 2011.

2.5.1 Initial Analysis of Defenses

Third-party cookie blocking. A standard defense against third-party web tracking is to block third-party cookies. This defense is insufficient for a number of reasons. First, different browsers implement third-party cookie blocking with different degrees of strictness. While Firefox blocks third-party cookies both from being *set* as well as from being *sent*, most other browsers (including Chrome, Safari, and Internet Explorer) only block the setting of third-party cookies. So, for example, Facebook can set a first-party cookie when the user visits `facebook.com`; in browsers other than Firefox, this cookie, once set, is available to Facebook from a third-party position (when embedded on another page).

Thus, in most browsers, third-party cookie blocking protects users only from trackers that are never visited directly — that is, it is effective against Behavior B (Vanilla) trackers but not against Behavior C (Forced) or Behavior E (Personal) trackers. Firefox's strict policy provides better protection, but at the expense of functionality like social widgets and buttons, some instantiations of OAuth or federated login, and other legitimate cross-domain behavior (thus prompting Mozilla to opt against making this setting the default [119]).

Do Not Track. The recently proposed Do Not Track header and legislation aim to give users a standardized way to opt out of web tracking. A browser setting (already implemented natively in Firefox, IE, and Safari) appends a DNT=1 header to outgoing requests, informing the receiving website that the user wishes to opt out of tracking. As of February 2012, Do Not Track is merely a policy technique that requires tracker compliance, providing no technical backing or enforcement. A major sticking point is the debate over the definition of tracking, as the conclusion of this debate determines to which parties the Do Not Track legislation will apply. As evidenced by the papers submitted to the W3C Workshop on Web Tracking and User Privacy,[4] many of the parties involved in tracking argue that their behaviors should not be considered tracking for the purposes of DNT. It is our hope that the tracking classification framework that we have developed and proposed in this paper can be used to further the discussion of what should be considered tracking in the policy

[4] `http://www.w3.org/2011/track-privacy/`

realm, and that a tool like TrackingTracker can be used in the browser to enforce and detect violations of Do Not Track.

Clearing client-side state. There has been some concern [180] that pervasive opt-out of tracking will create a tiered or divided web, in which visitors who opt out of tracking (via the DNT header or other methods) will not be provided with the same content as other visitors. One possible solution (also identified in [74]) to avoid tracking in the face of this concern is to constantly clear the browser's client-side state, regularly receiving new identifiers from trackers. This may be a sufficient solution for Behavior B, Behavior C, and Behavior D trackers, but it cannot protect users against Behavior E trackers to which they have identified themselves as a particular account holder (and thus logging back in will re-identify the same user). It is also hard to implement against Behavior A trackers, as they set first-party state on the websites that embed them, and it is difficult to distinguish in a robust manner the first-party state needed by the website from the state used by the Behavior A tracker. Other work [179] shows furthermore that fingerprinting techniques can re-identify a large fraction of hosts with fresh cookies.

Blocking popups. Most browsers today block popups by default, potentially making it more difficult for Behavior C trackers to maneuver themselves into first-party positions. However, websites can still open popups in response to user clicks. Furthermore, popups are only one way that Behavior C trackers can force a user to visit their site directly (and the easiest of these to detect and block). Other methods include redirecting the user's browser to the tracker's domain and back using javascript, or business relationships between the tracker and the embedding site that involve the site redirecting directly to a full-page interstitial ad controlled by the tracker's domain. These behaviors are hard or impossible to block as they are used throughout the web for other legitimate purposes.

Recent findings (February 2012) [105] furthermore revealed programmatic form submission as a new technique for Behavior C tracking in Safari, which treats form submissions as first-party interactions.

Private browsing mode. Private browsing mode, as explored in depth in [6], does not

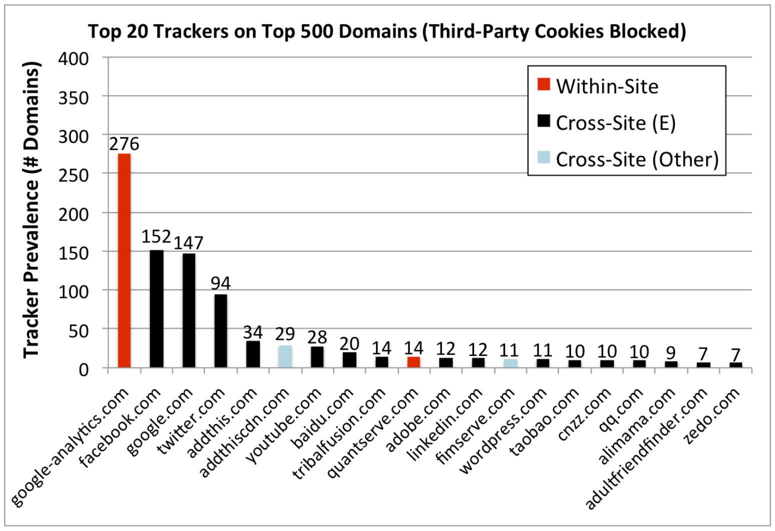


Figure 2.9: **Tracker Prevalence on Top 500 Domains (3rd Party Cookies Blocked).** Trackers are counted on domains, that is, if a particular tracker appears on two pages of a domain, it is counted once.

primarily address the threat model of web tracking. Instead, private browsing mode aims to protect browser state from adversaries with physical access to the machine. While the clearing of cookies when exiting private browsing mode can help increase a user's privacy in the face of tracking, private browsing mode does not aim to keep a user's browsing history private from remote servers.

2.5.2 Empirical Analysis of Defenses

As a part of our measurement study, we empirically analyzed the effectiveness of popup blocking and third-party cookie blocking to prevent tracking. The results of these measurements (run using the Alexa Top 500 domains, with 4 random links chosen from each) are summarized on the righthand side of Table 2.3. Overall, we find that existing defenses protect against a large portion of tracking, with the notable exception of Behavior E. We dive into the effects of each measured defense in turn.

With popups blocked, we did not observe significant differences in the tracking capabilities of most trackers. As expected, we observe fewer trackers exhibiting Behavior C

(Forced) — however, Behavior C using redirects remains, leaving three trackers exhibiting such behavior. We find most Behavior C trackers exhibit this type of behavior only occasionally, acting as Behavior B (Vanilla) the rest of the time. Thus, with third-party cookies enabled, popup blocking does not affect the capabilities of most trackers. Indeed, we suspect that most popups are used to better capture the user's attention rather than to maneuver the tracker domain into a first-party position. Nevertheless, this technique is sometimes used for this purpose.[5]

Third-party cookie blocking is, as expected, a better defense against tracking. However, recall that in most browsers other than Firefox, third-party cookie blocking only blocks the setting, not the sending, of cookies. Thus, if a tracker can ever set a cookie (via Behavior C or Behavior E), this cookie is available from that point forward. In Table 2.3, we distinguish the results for Firefox's strict cookie blocking policy: any type with an asterisk in the "Cookies Blocked" column disappears in Firefox (trackers that exhibit both Behavior A and E reduce to only A; the others disappear). Note the presence of one Behavior B tracker: this is `meebo.com`, which sets unique identifiers in LocalStorage in addition to browser cookies, leaving it unaffected by cookie blocking. Figure 2.9 shows the top 20 trackers for this measurement (compare to Figure 2.6), in which it is evident that most cross-site trackers have disappeared from the top 20, leaving the prominence of Behavior E trackers like Facebook, Twitter, YouTube, and others.

We also measured the effectiveness of disabling JavaScript, the most blunt defense against tracking. We find that it is extremely effective at preventing tracking behaviors that require API access to cookies to leak them, as is the case for Behavior A (Analytics) and Behavior D (Referred). However, trackers can still set cookies via HTTP headers and Behavior C trackers can use HTML redirects. Any tracking that requires only that content be requested from a tracker is not impacted — thus, while the scripts of Behavior E and other trackers cannot run (e.g., to render a social widget), they can be requested, thereby enabling tracking. Some trackers simply use `<noscript>` tags to fetch single-pixel images ("beacons") when more complex scripting techniques are not available. Despite being the

[5]`http://stackoverflow.com/questions/465662/`

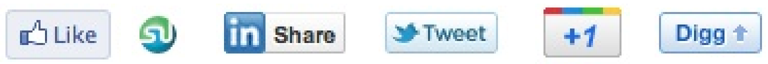


Figure 2.10: **Example Social Widgets.** Behavior E trackers expose social widgets that can be used to track users across all the sites on which these widgets are embedded.

most effective single defense, disabling JavaScript renders much of today's web unusable, making it an unworkable option for most users.

The Do Not Track header does not yet appear to have a significant effect on tracking, as evidenced by the sustained prevalence of most trackers in Table 2.3. Note that, as in all the results we report, we did exclude any trackers that set only non-unique and/or session cookies, as some trackers may respond to the DNT header by setting an opt-out cookie. We did notice that a few fairly prevalent trackers appeared to respond to the header, including `gemius.pl`, `serving-sys.com`, `media6degrees.com` and `bluekai.com`. These results are consistent with a recent set of case studies of DNT compliance [103].

Finally, we note that we did not observe any trackers actively changing behavior in an attempt to circumvent the tested defenses — that is, we did not observe more LocalStorage or more Flash cookies. Though we have not verified whether or not these trackers instead use more exotic storage mechanisms like cache Etags, we hypothesize that enough users do not enable these defenses to mobilize trackers to substantially change their behavior, and hence fall outside our common case explorations.

2.5.3 A New Defense: ShareMeNot

From these measurements, we conclude that a combination of defenses can be employed to protect against a large set of trackers. However, Behavior E trackers like Facebook, Google, and Twitter remain largely unaffected. Recall that these trackers can track logged-in users non-anonymously across any sites that embed so-called "social widgets" exposed by the tracker. For example, Facebook allows other websites to embed a "Like" button, Google exposes a "+1" button, and so on (see Figure 2.10 for a number of examples). These buttons present a privacy risk for users because they track users even when they choose not to click

Tracker	*Without ShareMeNot*	*With ShareMeNot*
Facebook	154	9
Google	149	15
Twitter	93	0
AddThis	34	0
YouTube	30	0
LinkedIn	22	0
Digg	8	0
Stumbleupon	6	0

Table 2.4: **Effectiveness of ShareMeNot.** ShareMeNot drastically reduces the occurrences of tracking behavior by the supported set of Behavior E trackers.

on any of the buttons.

When users can protect themselves from tracking in this fashion, it comes at the expense of the functionality of the button. In Firefox, the stricter third-party cookie blocking policy renders the buttons unusable. Other existing defenses, including the popular Disconnect browser extension,[6] work by simply blocking the tracker's scripts and their associated buttons from being loaded by the browser at all, thereby effectively removing the buttons from the user's web experience entirely.

We introduce ShareMeNot, an add-on for Chrome and Firefox that aims to find a middle ground between allowing the buttons to track users wherever they appear and retaining the functionality of the buttons when users explicitly choose to interact with them. It does this in one of two ways:

1. In one mode, ShareMeNot strips cookies from third-party requests to any of the supported Behavior E trackers under ordinary circumstances (as well as from any other blacklisted requests that are made in the context of loading such a button); when it detects that a user has clicked on a button, it allows the cookies to be included with the request, thereby allowing the button click to function as normal, transparently to the user.

2. In another mode, ShareMeNot blocks blacklisted requests completely, never loading any social media widgets. Instead, it replaces these widgets with client-side replace-

[6] `http://disconnect.me`

ment buttons; when a replacement button is clicked, ShareMeNot either dispatches the sharing request to the appropriate domain, or reloads the real social widget (in the case that the sharing request requires a CSRF token loaded with the widget, as in the case of Facebook and Google).

The second approach is more privacy-preserving, since no requests are made to the tracker's domain until the user clicks a widget, but can have unintended effects on webpage layout. Thus, we give ShareMeNot users the option to select their preferred mode.

The use of ShareMeNot shrinks the profile that the supported Behavior E trackers can create to only those sites on which the user explicitly clicks on one of the buttons — at which point the button provider must necessarily know the user's identity and the identity of the site on which the button was found in order to link the "like" or the "+1" action to the user's profile. No other existing approach can both shrink the profile a Behavior E tracker can create while also retaining the functionality of the buttons, though concurrent work on the Priv3 Firefox add-on [34] adopts the same basic approach; as of May 2014, Priv3 supports fewer widgets, is no longer actively maintained, and, to our knowledge, was not iteratively refined through measurement.

We experimentally verified the effectiveness of ShareMeNot on Firefox. As summarized in Table 2.4, ShareMeNot dramatically reduces the presence of the Behavior E trackers it supports to date. We chose to support these sites based in part on our initial, pre-experimental perceptions of popular third-party trackers, and in part based on our experimental discovery of the top trackers. ShareMeNot entirely eliminates tracking by most of these, including Twitter, AddThis, Youtube, Digg, and Stumbleupon. While it does not entirely remove the presence of Facebook and Google, it reduces their prevalence to 9 and 15 occurrences, respectively. In the Facebook case, this is due to the Facebook comments widget, which triggers additional first-party requests (containing tracking information) not blacklisted by ShareMeNot; the Google cases appear mostly on other Google domains (e.g., `google.ca`).

ShareMeNot is available at `http://sharemenot.cs.washington.edu/`, and its implementation is described in more detail in a 2012 article [141]. As of June 2014, ShareMeNot has over 16,000 active users on Firefox and Chrome. Its techniques have been adopted by Ghostery, another major tracking-related browser extension with over 2.3 million users on Firefox and Chrome. Additionally, as of June 2014, I am working with the Electronic Frontier Foundation (EFF), a nonprofit digital rights organization, to incorporate ShareMeNot into its new anti-tracking add-on, Privacy Badger.[7]

2.6 Platform for Tracking Detection, Blocking, Measurement

Following our initial measurement work using TrackingTracker [139], we refined and extended the original measurement tool into a browser-based platform for tracking detection, blocking, and measurement: TrackingObserver. We describe this platform, implemented as a Chrome extension available at `http://trackingobserver.cs.washington.edu`, in this section. The TrackingObserver platform:

1. dynamically detects trackers based on their client-side (cookie-related) behavior, and
2. provides APIs that expose the collected data along with blocking and measurement functionality.

2.6.1 The Need for a Web Tracking Detection Platform

Though web tracking has received a great deal of attention, prompting both research, policy, and commercial solutions, the solution space is fragmented, requiring each new implementation to begin from scratch. We observe that most available tools — including tools like Ghostery, Collusion (renamed Lightbeam[8]), Disconnect, ShareMeNot, and others — innovate not in how they *detect* trackers, but in how they *handle* trackers once detected. We thus argue that innovation in web tracking tools would benefit from a shared platform that provides the tracker detection functionality and that exposes the necessary APIs to allow for innovation on top of this foundation.

We further observe that nearly all existing tools that detect and/or block third-party trackers do so using a blacklist model, in which the tool relies on a predefined (possibly

[7]`https://www.eff.org/privacybadger`

[8]`http://www.mozilla.org/en-US/lightbeam/`

user-loaded) list of known tracker domains. Though this model is simple to implement, it suffers from several drawbacks:

- Blacklists must be kept complete and up-to-date. Prior work examining the effectiveness of anti-tracking tools has shown that blacklists are often incomplete [104].
- Users must trust the creator of the blacklists not to exclude tracker domains with whom the blacklist creator is colluding, or to include non-tracker domains (e.g., in order to block a competitor's site). This is a realistic concern: the same prior work mentioned above [104] revealed that some tracking list providers intentionally override others to allow certain trackers. As another example of complicated incentives, Ghostery is owned by a company that helps businesses use tracking effectively [151].
- As discussed earlier in this chapter, tracking behavior is complex and differs not only between trackers but between different instances of a given tracker. Blacklist-based solutions generally do not differentiate different types of trackers, and certainly do not differentiate different behaviors of a given tracker.

Thus, we argue that web trackers should not be detected (only) using a blacklist, but rather by observing and categorizing client-side behavior. That is, tracking behavior should be detected dynamically within the browser, by observing third-party requests and client-side state (such as cookies) in real-time.

Dynamic detection addresses the issues with blacklists that we described above. First, dynamic tracking detection is neutral, based on tracking mechanisms and not on a potentially ad-hoc (or worse, biased) list of known trackers. The dynamic detection code can be audited once (or infrequently), rather than on every update to the blacklist. Second, because dynamic detection is based on behaviors, it can detect new trackers automatically, without requiring any intervention or extra steps on the part of the anti-tracking tool developer or maintainer.

Dynamic detection can also be combined with a blacklist approach by reporting back dynamically-detected trackers from individual users and combining this information to create a global list of known trackers. (Prior work [104] has observed that community maintained blacklists are most effective.) It can also be combined with the more traditional blacklist-based approach of known tracker domains.

Finally, based on our experience with TrackingTracker (our original measurement tool), we learned that the implementation this dynamic detection approach is more complex than a blacklist-based implementation. It is therefore more difficult for each tool to adopt individually. The TrackingObserver platform which we released provides this behavior-based tracking detection algorithm for use by other developers. We hope that this platform will benefit both researchers wishing to gather data about the web tracking ecosystem or experiment with uses of this data, as well as commercial solutions wishing to help users manage their privacy.

2.6.2 TrackingObserver's APIs

TrackingObserver serves as a platform that provides dynamic tracking detection functionality and exposes a set of APIs to add-ons, themselves implemented as Chrome extensions. Table 2.5 summarizes the APIs provided by the TrackingObserver platform. These APIs provide several types of functionality:

1. *Plumbing:* These APIs set up links between add-ons and the TrackingObserver platform itself.
2. *Data:* These APIs return data about trackers detected by TrackingObserver.
3. *Blocking:* These APIs support functionality for blocking and unblocking trackers.
4. *Measurement:* These APIs support automated measurement of web tracking.

We chose to provide these APIs based on the functionality supported by existing tracking-related browser extensions, and iteratively based on our own experience creating a number of add-ons to TrackingObserver (described below).

2.6.3 Case Studies: Add-Ons to TrackingObserver

To evaluate the power of TrackingObserver as a platform, we implemented a number of case study add-ons on top of the TrackingObserver platform APIs described above. These add-ons are depicted in Figures 2.11–2.14 and provide visualization, blocking, and measurement functionality. Each add-on is itself a full-fledged Chrome extension that uses Chrome's cross-

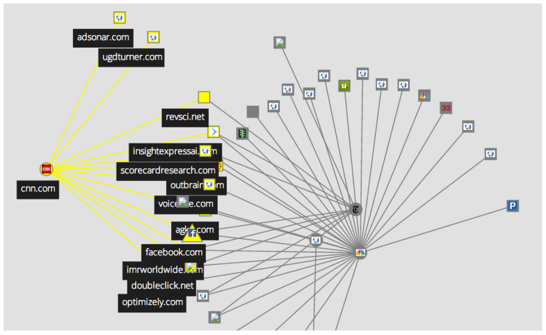
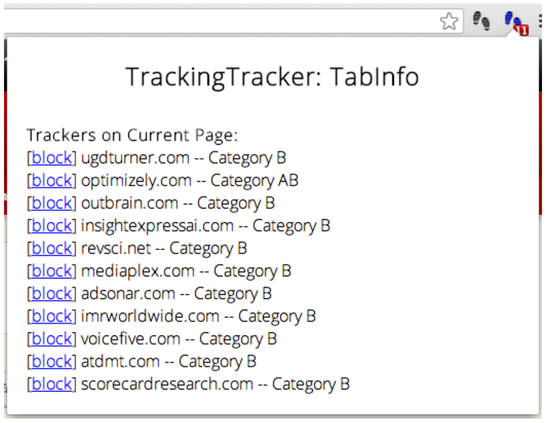


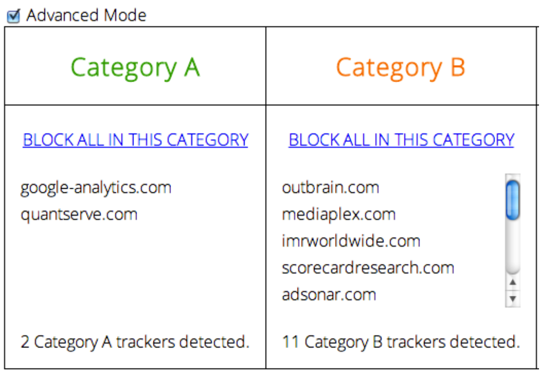
Figure 2.11: **Graph Visualization.** This add-on shows a graph of trackers and visited sites (similar to Collusion/Lightbeam).



TrackingTracker: TabInfo

Trackers on Current Page:
[block] ugdturner.com -- Category B
[block] optimizely.com -- Category AB
[block] outbrain.com -- Category B
[block] insightexpressai.com -- Category B
[block] revsci.net -- Category B
[block] mediaplex.com -- Category B
[block] adsonar.com -- Category B
[block] imrworldwide.com -- Category B
[block] voicefive.com -- Category B
[block] atdmt.com -- Category B
[block] scorecardresearch.com -- Category B

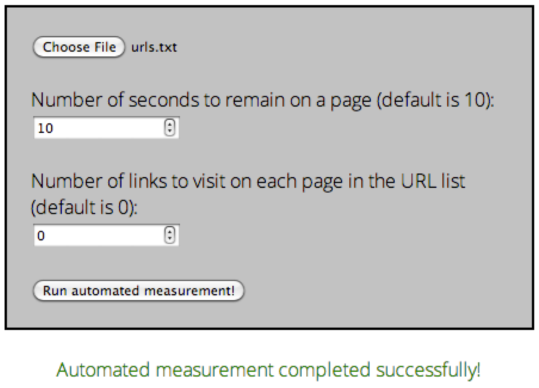
Figure 2.12: **Per-Tab Visualization.** This add-on displays a list of the trackers detected on the current tab (similar to Ghostery), along with their tracking category (or categories) and a block/unblock link.



☑ Advanced Mode

Category A	Category B
BLOCK ALL IN THIS CATEGORY	BLOCK ALL IN THIS CATEGORY
google-analytics.com quantserve.com	outbrain.com mediaplex.com imrworldwide.com scorecardresearch.com adsonar.com
2 Category A trackers detected.	11 Category B trackers detected.

Figure 2.13: **Blocking Dashboard.** This add-on allows users to view and manage blocked trackers and categories.



Choose File urls.txt

Number of seconds to remain on a page (default is 10):
10

Number of links to visit on each page in the URL list (default is 0):
0

Run automated measurement!

Automated measurement completed successfully!

Figure 2.14: **Measurement Tool.** This add-on helps researchers perform automated measurements of web tracking.

API Name	*Functionality*	*Type*
registerAddon(name, link)	Registers an add-on to the TrackingObserver platform.	Plumbing
registerForCallback()	Registers the add-on for callbacks when tracking is detected.	Plumbing
getTrackers()	Returns information (domain, categoryList, trackedSites) about all logged trackers.	Data
getTrackersOnCurrentTab()	Returns information about trackers logged on the most recent load of the active tab.	Data
getTrackersBySite()	Returns information about trackers indexed by site on which they were logged.	Data
blockTrackerDomain(domain)	Causes TrackingObserver to start blocking third-party requests to this domain.	Blocking
unblockTrackerDomain(domain)	Causes TrackingObserver to stop blocking third-party requests to this domain.	Blocking
trackerDomainBlocked(domain)	Returns whether or not the given domain is currently on the block list.	Blocking
getBlockedDomains()	Returns the full set of currently blocked domains.	Blocking
blockCategory(category)	Causes TrackingObserver to start blocking all trackers of the given category.	Blocking
unblockCategory(category)	Causes TrackingObserver to stop blocking all trackers of the given category.	Blocking
trackerCategoryBlocked(category)	Returns whether or not the entire given category is blocked.	Blocked
browseAutomatically(urls)	Drives the browser to visit a list of URLs automatically.	Measurement

Table 2.5: **TrackingObserver Platform APIs.** The TrackingObserver platform implements the APIs described here and makes them accessible to platform add-ons (Section 2.6.3). Both TrackingObserver and its add-ons are implemented as Chrome extensions.

extension message passing support to call TrackingObserver's APIs. Table 2.6 summarizes these add-ons, along with the lines of JavaScript needed to implement each one. The implementation complexity of each add-on is small, requiring only between 70 and 350 lines of JavaScript and relying to the TrackingObserver platform (which itself requires 858 lines of JavaScript) to do the more complex tracking detection work.

2.7 Related Work

Web tracking has been studied extensively in prior, concurrent, and subsequent work to the work described in this chapter. For example, Mayer et al. provide a good recent summary of the state of web tracking technology and policy [107]. A number of studies have empirically

Add-On	*Functionality*	*Lines of Code (JavaScript only)*
Graph Visualization	Graphs all trackers encountered while browsing (based on Collusion).	344
Per-Tab Visualization	Displays information about and allows (un)blocking of trackers on current tab (based on Ghostery).	106
Blocking Dashboard	Shows which trackers are blocked and allows (un)blocking by domain and by tracking category.	190
Raw Data Display	Shows all collected data in a raw form, suitable for further processing (e.g., using Excel).	98
Measurement Tool	Run an automated web tracking measurement.	74

Table 2.6: **TrackingObserver Add-On Case Studies.** This table describes the add-ons to TrackingObserver that we implemented to demonstrate and evaluate the capabilities enabled by its APIs. The functionality of existing tracking-related browser extensions (e.g., Collusion/Lightbeam, Ghostery) can be supported by TrackingObserver. Each add-on required a small amount of implementation effort given the common tracking detection functionality provided by TrackingObserver.

examined tracking on the web (e.g., [74, 87, 88]) and on mobile devices (e.g., [45, 64]). Especially in early work in this space [87], researchers frequently did not distinguish between different types of trackers, grouping together, for example, Google Analytics (a within-site tracker) and Doubleclick (a cross-site tracker), though they touch on aspects in prior work [88]. As discussed, we believe that this distinction is fundamentally important for understanding and responding to web tracking.

A five-year study of modern web traffic [73] found that ad network traffic accounts for a growing percentage of total requests (12% in 2010). They find Google Analytics on up to 40% of the pages reflected in their data, a number that has increased to over 50% in our data. Another measurement study of web tracking appeared in [79], in which the authors examined the prevalence of cookie usage and P3P policies. Further afield, studies have considered other instances of personal information leakage between websites (e.g., [77, 78, 85, 86]).

Much web tracking relies on client-side storage of unique identifiers (e.g., using browser cookies), and some trackers have been found explicitly circumventing user intent by "respawning" unique tracking identifiers after the user deletes them (e.g., repopulating browser cookies from Flash cookies) [12, 153]. Browser and machine fingerprinting techniques, which require that the tracker store no explicit state in the user's browser, have also recently received increased attention [3, 39, 82, 122, 179].

From the user perspective, studies have shown that users are concerned by web tracking practices (e.g., [91, 107, 108, 167]). Efforts like Do Not Track [103, 106] and proposed legislation have arisen to address the issue from the policy side. Unfortunately, Do Not Track talks appear to have stalled [152], indicating that a stronger technology solution may after all be necessary to provide users with effective prevention capabilities.

Technical approaches to preventing links between a user's browsing behaviors include a variety of browser extensions (such as Ghostery, Collusion/Lightbeam, and others) and browser settings (such as third-party cookie blocking), whose effectiveness has been studied in this chapter and elsewhere [104]. More heavyweight approaches include the Tor Browser Bundle[9] and recent research on unlinkable pseudonyms based on IPv6 addresses [65]. Researchers have also experimented with using machine learning techniques to prevent tracking [16]. Finally, researchers have focused on designing privacy-preserving alternatives to tracking-related functionality — including advertising (e.g., [52, 62, 166]), analytics (e.g., [7]) and social media widgets (e.g. [34, 83]).

Additionally, there have been significant online discussions about tracking, e.g., [103]. Finally, entire workshops and working groups on tracking have emerged, e.g., the 2011 Workshop on Internet Tracking, Advertising, and Privacy, the 2011 W3C Workshop on Web Tracking and User Privacy, and the W3C Tracking Protection Working Group.

2.8 Summary

In this chapter we presented an in-depth empirical investigation of third-party web tracking. Our empirical investigation builds on the introduction of what we believe to be the first comprehensive classification framework for web tracking based on client-side observable behaviors. We believe that this framework can serve as a foundation for future technical

[9]`https://www.torproject.org/projects/torbrowser.html.en`

and policy initiatives. We additionally evaluated a set of common defenses on a large scale and observed a gap — the ability to defend against Behavior E tracking with social media widgets, like the Facebook “Like” button, while still allowing those widgets to be useful. In response, we developed and evaluated ShareMeNot, which is designed to thwart such tracking while still allowing the widgets to be used. Finally, we expanded our original measurement tool into TrackingObserver, a web tracking detection, measurement, and blocking platform. This work thereby tackles the first key challenge identified in Chapter 1, embedded third-party principals, in the context of the web.

Chapter 3

USER-DRIVEN ACCESS CONTROL: RETHINKING PERMISSION GRANTING FOR MODERN OPERATING SYSTEMS

This chapter considers another key challenge identified in Chapter 1: the problem of application permission granting in modern operating systems, such as iOS, Android, Windows Phone, Windows 8, and web browsers. Rather than asking users to explicitly make access control decisions (e.g., by responding to a permission prompt), we take the approach of *user-driven access control*, whereby permission granting is built into existing user actions in the context of an application. To allow the system to precisely capture permission-granting intent in an application's context, we introduce *access control gadgets* (ACGs), which leverage an application's ability to embed user interface content from third-party principal — in this case, a system-trusted principal. This work was first described in a 2012 paper [138].

3.1 Motivation and Overview

Many modern client platforms treat applications as distinct, untrusted principals. For example, smartphone operating systems like Android [58] and iOS [9] isolate applications into separate processes with different user IDs, and web browsers implement the same-origin policy [169], which isolates one web site (or application) from another. By default, these principals receive limited privileges; they cannot, for example, access arbitrary devices or a global file system. From a security perspective, this is an improvement over desktop systems, which treat users as principals and grant applications unrestricted resource access by virtue of installation.

Unfortunately, these systems provide inadequate functionality and security. From a functionality standpoint, isolation inhibits the client-side manipulation of user data across applications. For example, web site isolation makes it difficult to share photos between two sites (e.g., Picasa and Flickr) without manually downloading and re-uploading them. While applications can pre-negotiate data exchanges through IPC channels or other APIs, requiring every pair of applications to pre-negotiate is inefficient or impossible. From a security standpoint, existing access control mechanisms tend to be coarse-grained, abrasive, or inadequate. For instance, they require users to make out-of-context, uninformed decisions at install time via manifests [13, 58], or they unintelligently prompt users to determine their intent [9, 112].

Thus, a pressing open problem is how to allow users to grant applications access to *user-owned resources*: privacy-and cost-sensitive devices and sensors (e.g., the camera, GPS, or SMS), system services and settings (e.g., the contact list or clipboard), and user content stored with various applications (e.g., photos or documents). To address this problem, we advocate *user-driven access control*, whereby the system captures user intent via authentic user actions in the context of applications. Prior work [116, 149, 157] applied this principle largely in the context of least-privilege file picking, where an application can access only user-selected files; in this chapter, we *generalize* it for access to all user-owned resources.

Furthermore, we introduce *access control gadgets* (ACGs) as an operating system technique to capture user intent. Each user-owned resource exposes user interface (UI) elements called ACGs for applications to embed. The user's authentic UI interactions with an ACG grant the embedding application permission to access the corresponding resource. Our design ensures the display integrity of ACGs, resists tampering by malicious applications, and allows the system to translate authentic user input on the ACGs into least-privilege permissions.

The ACG mechanism enables a permission granting system that *minimizes unintended access* by letting users grant permissions at the time of use (unlike the manifest model) and *minimizes the burden on users* by implicitly extracting a user's access-control intentions from his or her in-application actions and tasks (rather than, e.g., via prompts).

In addition to system-controlled resources, we generalize the notion of user-driven access control to sensitive resources controlled by applications with *application-specific ACGs.* Applications (e.g., Facebook) that provide APIs with privacy or cost implications (e.g., allowing access to friend information) can expose ACGs to require authentic user actions before allowing another application to invoke those APIs.

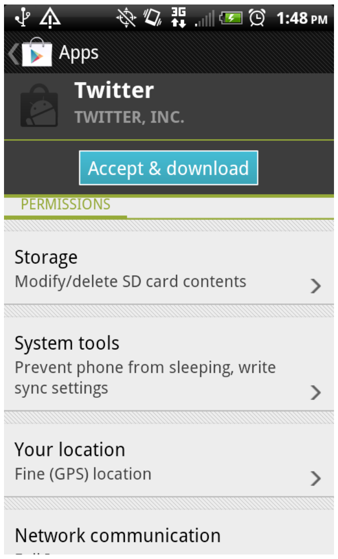
We built a prototype of user-driven access control into an existing research system [173] that isolates applications based on the same-origin policy. A quantitative security evaluation shows that our system prevents a large swath of user-resource vulnerabilities. A study of today's Android applications shows that our design presents reasonable tradeoffs for developers. Finally, we conduct two different user studies (with 139 and 186 users, respectively); the results indicate that user-driven access control better matches user expectations than do today's systems.

3.2 State of the Art in Permission Granting

We motivate our work by identifying shortcomings in existing permission granting systems. Here, we discuss common and commercially available approaches; we discuss additional related research approach in Section 3.7.

Global resources. Traditional desktop systems expose system resources to applications simply by globalizing them: applications have unfettered access to a global file system, to the system clipboard, and to peripheral devices. Similarly, smartphone OSes expose a global clipboard to applications. While user-friendly in the benign case, this model allows applications to access resources even when they don't need them (i.e., it violates the principle of least-privilege) and allows unintended access (e.g., attacks that pollute the globally accessible clipboard [37]). Our user studies (Section 3.6.2) indicate that such exposures contradict users' expectations (e.g., users expect data on the clipboard to remain private until pasted).

Manifests. Applications on Android, Facebook, Windows Phone, and Window 8 use install-time manifests to request access to system resources. Figure 3.1 shows an example of an Android manifest. Once the user agrees, the installed application receives permanent access to the requested resources. This violates least-privilege, as installed applications may access these resources at any time, not just when needed to provide intended functionality (e.g., allowing malicious applications to surreptitiously take photos [161]). Furthermore, studies indicate that many applications ask for more permissions than needed [15, 49] and that users pay little attention to, and show little comprehension of, Android manifests [50]. Recent Android malware outbreaks suggest that manifests do not prevent users from in-


1:48 PM

Apps

Twitter

TWITTER, INC.

Storage

Modify/delete SD card contents

System tools

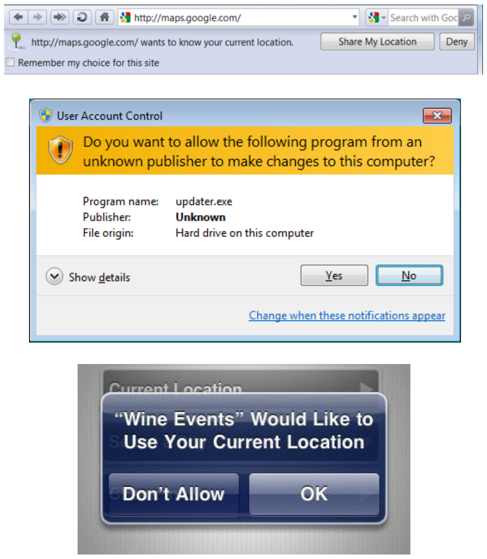
Prevent phone from sleeping, write sync settings

Your location

Fine (GPS) location

Network communication

Figure 3.1: **Android Manifest.** Android, Facebook, Windows Phone, and Window 8 use install-time manifests to grant permissions. Manifests over-privilege applications by granting permanent access to requested resources, and user comprehension of manifests is low.


http://maps.google.com/

http://maps.google.com/ wants to know your current location.

Remember my choice for this site

User Account Control

Do you want to allow the following program from an unknown publisher to make changes to this computer?

Program name: updater.exe

Publisher: Unknown

File origin: Hard drive on this computer

Current Location

“Wine Events” Would Like to Use Your Current Location

Figure 3.2: **Prompts.** Browsers (top) and iOS (bottom) prompt for geolocation access; Windows (center) prompts for actions requiring administrative privilege. Prompts attempt to verify user intent, but their disruptiveness teaches users to ignore and click through them.

stalling applications that ask for excessive permissions [13].

While we argue against user-facing manifests, manifests are a useful mechanism for communication between an application and the system; an application can specify the maximum set of permissions it may need, and the system can then restrict the application accordingly to mitigate the effects of an application compromise. In this way, system-facing manifests may be applied in conjunction with user-driven access control.

Prompts. By contrast, iOS [9] prompts users the first time an application wishes to access a resource. Windows displays a User Account Control prompt [112] when an application requires additional privileges to alter the system. Nascent support for user-owned resources in HTML5 involves prompting for geolocation access. Figure 3.2 shows examples of such prompts.

While these prompts attempt to verify user intent, in practice, the burden they place on users undermines their usefulness. Specifically, when the user intends to grant access, the prompts seem unnecessary, teaching users to ignore them [178]. Several studies have shown that users ignore interruptive warnings [56] and click through even illegitimate consent prompts [118]. As a result, systems like iOS prompt users only the first time an application attempts to access a resource, and allow the application unrestricted access to that resource thereafter (violating least-privilege).

No access. To address security concerns, some systems simply do not support application access to system resources. For example, in today's browsers [135], web applications generally cannot access a user's local devices and have restricted access to the file system and to the clipboard [113]. (Browser plugins, which have access to all system resources, are a dangerous exception.) Research browsers and browser operating systems [31, 61, 69, 159, 172, 173] also have not addressed access control for system resources.

The functionality needs of legitimate applications are clearly at odds with this restrictive model and have led web developers to circumvent such restrictions using browser plugins. For instance, developers have created copy-and-paste buttons by overlaying transparent Flash elements and tricking users into clicking on them (e.g., via clickjacking) [123]. Browsers have thus begun to expose more resources to web applications, as I discuss further below. Existing specifications for permitting access to system resources [171] note only that permission-granting should be tied to explicit user actions, but they do not address how these should be mapped to system-level access control decisions.

3.3 Goals and Context

We consider the problem of allowing a system to accurately capture a user's access control decisions for user-owned resources. Learning from existing systems, our goals are to enable permission granting that is (1) in-context (unlike manifests), (2) non-disruptive (unlike system prompts), and (3) least-privilege (unlike most previous systems).

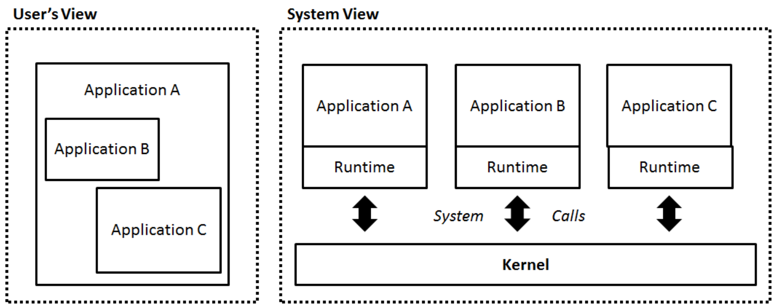


Figure 3.3: **Application Model.** As in web mashups, applications may embed other applications. Each application is isolated and runs atop a generic runtime (e.g., a browser renderer or Win32 library).

3.3.1 System Model

We assume (see Figure 3.3) that applications are isolated from each other according to some principal definition (such as the same-origin policy), share no resources, and have no access to user-owned resources by default. Applications may, however, communicate via IPC channels. Some existing systems (e.g., browsers and smartphones) support many of these features and could be modified to support the rest.

We further assume that applications (and their associated principals) may embed other applications (principals). For instance, a news application may embed an advertisement. Like all applications, embedded principals are isolated from one another and from the outer embedding principal.[1] We assume that the kernel has complete control of the display and that applications cannot draw outside of the screen space designated for them. An application may overlap, resize, and move embedded applications, but it cannot access an embedded application's pixels (and vice versa) [147, 172]. Furthermore, the kernel dispatches UI events only to the application with which the user is interacting. The prototype system [173] that we use to implement and evaluate user-driven access control supports these properties; Chapter 4 discusses how a system can achieve secure embedded user interfaces

[1]Research browser Gazelle [172] advocated this isolation. Commercial browsers like Chrome and IE have not yet achieved it, instead putting all principals embedded on one web page into the same OS process. Today's smartphones do not yet support cross-principal content embedding, but we anticipate this to change in the future; Chapter 4 discusses how it can be achieved securely.

in general.

To provide access control for user-owned resources, a system must support both access control *mechanisms* and access control *policies*. For the former, to simplify the discussion, we assume that for each user-owned resource, there is a set of system APIs that perform privileged operations over that resource. The central question of this work is how to specify policies for these mechanisms in a user-driven fashion.

3.3.2 User-Owned Resources

In this work, we study access control for user-owned resources. Specifically, we assume that by virtue of being installed or accessed, an application is granted isolated access to basic execution resources, such as CPU time, memory, display, disk space, and network access.[2]

We consider all other resources to be user-owned, including: (1) Devices and sensors, both physical (e.g., microphone, GPS, printer, and the phone's calling and SMS capabilities) and virtual (e.g., clipboard and contacts), (2) user-controlled capabilities or settings, such as wiping or rebooting the device, and (3) content (e.g., photos, documents, or friends lists) residing in various applications.

3.3.3 Threat Model

We aim to achieve our above-stated goals in the face of the following threat model. We consider the attacker to be an untrusted application, but we assume that the kernel is trustworthy and uncompromised; hardening the kernel is an orthogonal problem (e.g., [132, 146]). We assume attacker-controlled applications have full network access and can communicate via IPC to other applications on the client side. We classify potential threats to user-owned resources into three classes:

1. **When:** An application accesses user-owned resources at a moment when the user did not intend.

[2]Network access is part of today's manifests. We believe that approving network access for safety is too much to ask for most users. Nevertheless, we allow an application to provide a non-user-facing manifest to the OS to enable better sandboxing (e.g., by restricting network communication to a whitelist of domains).

2. **Who:** An application other than the one intended by the user accesses user-owned resources.
3. **What:** An application grants another application access to content other than that specified by the user. This false content may be *malicious content* intended to exploit another application, and/or it may be a user's authentic content, but not the content to which the user intended to grant access (*leaked content*).

In this work, we restrict this model in two ways. First, we do not address the problem of users misidentifying a malicious application as a legitimate application. For example, a user may mistakenly grant camera access to a fake, malicious Facebook application. Principal identification is a complementary problem (even if solved, we must still address user-driven access control for well-identified principals). Second, we consider out of scope the problem of "what" data is accessed. Input sanitization to protect against malicious content is a separate research problem. Furthermore, applications possessing user data can already leak it via network and IPC communication. Techniques like information flow control [181] may help remove this assumption but are orthogonal.

In this work, we aim to raise the bar for malicious applications attempting to access user-owned resources without user authorization, and to reduce the scope of such accesses.

3.4 Design: User-Driven Access Control

The fundamental problem with previous permission granting models is that the system has no insight into the user's in-application behaviors. Consequently, the system cannot implicitly learn the user's permission-granting intent and must ask the user explicitly (e.g., via an out-of-context manifest or a system prompt uncoordinated with the user's in-application behavior). Our goal is to bridge this disconnect between a user's activities in an application and the system's permission management for sensitive resources. In particular, we aim to enable user-driven access control, which allows the user's natural UI actions in the context of an application to govern the access control of user-owned resources.

To achieve this goal, we need (1) applications to build permission-granting UIs into their own context, and (2) the system to obtain the user's authentic permission-granting intent from his or her interaction with these UIs. We introduce *access control gadgets* (ACGs) as

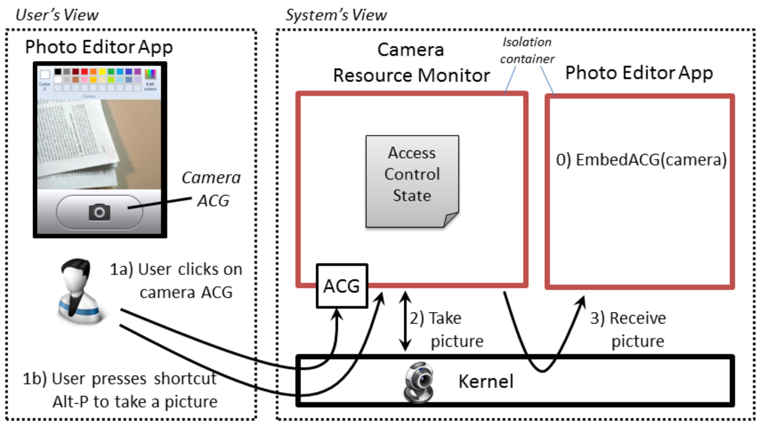


Figure 3.4: **System Overview.** Resource monitors mediate access to user-owned resources. They capture the user's intent to grant an application access to that resource via (1a) UI elements in the form of access control gadgets, and/or via (1b) permission-granting input sequences detected by the kernel.

this permission-granting UI, a key mechanism to achieve user-driven access control.

Figure 3.4 gives an overview of our system. Each type of user-owned resource (e.g., the abstract camera type) has a *user-driven resource monitor*. This resource monitor (RM) is a privileged application that exclusively mediates access to all physical devices (e.g., multiple physical cameras) of that resource type and manages the respective access control state. The RM exposes to applications (1) access control gadgets (ACGs) and (2) device access APIs. ACGs are permission-granting UI elements which applications can embed. A user's authentic interactions with the ACGs (e.g., arrow 1a in Figure 3.4) allow the RM to capture the user's permission-granting intent and to configure the access control state accordingly. For example, clicking on the camera ACG grants its host application permission to take a picture with the camera. Applications can invoke the device access APIs exposed by the RM only when granted permission.

Users can also grant permissions via user-issued *permission-granting input sequences.* For example, the camera RM may designate the Alt-P key combination or the voice command "Take a picture" as the global input sequence for taking a picture. Users can then use these

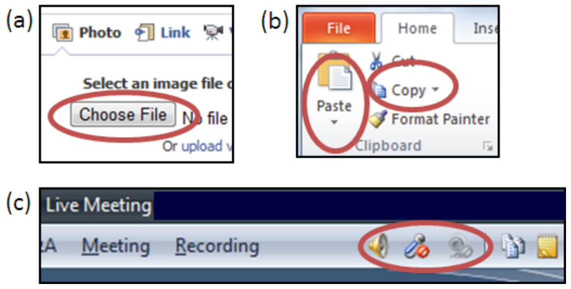


Figure 3.5: **Example Access Control Gadgets (ACGs).** The circled UI elements are application-specific today, but in our model, they would be ACGs for (a) content picking, (b) copy & paste, or (c) device access.

directly to grant the permission to take a picture (arrow 1b in Figure 3.4).

ACGs apply to a broad range of access control scenarios, including device access (e.g., camera and GPS), system service access (e.g., clipboard and wipe-device), on-device user data access (e.g., address book and autocomplete data), and application-specific user data access (e.g., Facebook's friends list). Figure 3.5 gives a few examples. We later describe our implementation of a representative set of ACGs in Section 3.5.3.

An RM (e.g., a camera RM) may manage multiple physical devices (e.g., multiple cameras). The RM provides a device discovery API so that an application can choose to employ a specific physical device. An RM can also incorporate new UI elements into its ACG(s) when a new device (e.g., a new camera with additional menu options) is plugged into the system. An RM must be endorsed (and signed) by the OS vendor and trusted to manage its resource type; new RMs are added to the system like device drivers (e.g., by coming bundled with new hardware or downloaded). In this paper, we will not further discuss these aspects of RM design and instead focus on authentically capturing and carrying out user intent.

In the rest of this section, we first present our design enabling the system to obtain *authentic* user interactions with an ACG (Section 3.4.1); we defer the design of specific ACGs to Section 3.5.3. Our system supports a variety of access semantics (Section 3.4.2), such as one-time, session, and permanent access grants; it also supports ACG composition

(Section 3.4.3) to reduce the number of user actions required for granting permissions. In Section 3.4.4, we generalize user-driven access control to allow all applications to act as RMs and expose ACGs for application-specific resources. Finally, Section 3.4.5 describes alternate ways in which users can grant permissions via kernel-recognized permission-granting input sequences that arrive via keyboard, mouse, or voice commands.

3.4.1 Capturing User Intent with ACGs

To accurately capture the user's permission-granting intent, the kernel must provide a trusted path [177] between an ACG and the user. In particular, it must ensure the integrity of the ACG's display and the authenticity of the user's input; we discuss these protections here, and develop a more general set of requirements and techniques for securing embedded user interfaces, including ACGs, in Chapter 4. Helping the user distinguish forged ACGs from real ACGs is less critical; an entirely forged ACG (i.e., not a real ACG with a forged UI) cannot grant the embedding application any permissions.

ACG→User: Ensuring Display Integrity

It is critical that the user sees an ACG before acting on it. We ensure the display integrity of ACGs by guaranteeing the following three properties.

1. *Display isolation*: The kernel enforces display isolation between different embedded principals (Section 3.3.1), ensuring that an application embedding an ACG cannot set the ACG's pixels.
2. *Complete visibility*: The kernel ensures that active ACGs are completely visible, so that malicious applications cannot overlay and manipulate an ACG's display to mislead users (e.g., overlaying labels to reverse the meaning of an ACG's copy/paste buttons). To this end, we require that ACGs be opaque and that the kernel only activate an ACG when it is entirely visible (i.e., at the top of the display's Z ordering). An ACG is deactivated (and greyed out) if any portion becomes obscured.
3. *Sufficient display duration*: The kernel ensures that the user has sufficient time to perceive an active ACG. Without this protection, applications can mount timing-

based clickjacking attacks in which an ACG appears just as the user is about to click in a predictable location. Thus, the kernel only activates an ACG after it has been fully visible in the *same* location for at least 200 ms, giving the user sufficient time to react (see [99]). We use a fade-in animation to convey the activation to the user, and temporarily deactivate an ACG if it moves. We postulate (as do the developers of Chrome [26] (Issue 52868) and Firefox [120] (Advisory 2008-08)) that the delay does not inconvenience the user in normal circumstances.

Customization. There is a tension between the consistency of an ACG's UI (that conveys permission-granting semantics to users) and the UI customization desired by developers to better build ACGs into application context. For example, developers may not wish to use the same geolocation ACG for different functions, such as "search nearby," "find deals," and "check in here." However, allowing applications to arbitrarily customize ACGs increases the attack surface of an RM and makes it easy for malicious applications to tamper with an ACG's display (e.g., a malicious application can disguise a gadget that grants geolocation access as one that does not).

With respect to customization, we consider three possible points in the design space.

1. *Disallow customization.* Disallowing customization ensures display integrity, but might incentivize developers to ask for permanent access unnecessarily so that they can design their desired UIs.
2. *Limited dynamic customization.* ACGs can be customized with parameters, such as color and size, with a fixed set of possible values for each parameter.
3. *Arbitrary customization with a review process.* Another possibility is to allow arbitrary customization but require explicit review and approval of custom ACGs (similar to how applications are reviewed in today's "app stores"). In this model, application developers use the available ACGs by default, but may submit customized ACGs for approval to an "ACG store." This option incentivizes developers to use the standard non-customized ACGs in the common case, and ensures display integrity for the approved ACGs that are submitted to the "store."

In our evaluation of existing Android applications (Section 3.6.7), we find that gadgets with limited customization would be sufficient in most cases.

User→ACG: Interpreting Authentic User Input

In addition to guaranteeing the display integrity of ACGs as described above, we must ensure (1) that input events on ACGs come authentically from the user, and (2) that the kernels grants permissions to the correct application.

For (1), our kernel enforces input event isolation, i.e., the kernel dispatches user input events originating from physical devices (e.g., keyboard, touchscreen, mouse, or microphone) only to the in-focus application. No application can artificially generate user input for other applications. Thus, when an ACG is in focus and the user acts on it, the user's input, dispatched to the corresponding RM, is guaranteed to be authentic. Equally important, the kernel must protect input-related feedback on the screen. For example, the kernel controls the display of the cursor and ensures that a kernel-provided cursor is shown when the mouse hovers over a gadget. This prevents a malicious application from confusing the user about where they are clicking within an ACG.

For (2), it is straightforward for the kernel to determine the correct application to which to grant permissions when there is a single level of embedding, i.e., when a top-level application embeds an ACG. However, care must be taken for multi-level embedding scenarios. For example, a top-level application may embed another application which in turn embeds an ACG. Such application nesting is a common pattern in today's web and can be arbitrarily deep.

Thus, the kernel must prevent an embedded application from tricking users into believing that the ACG it embeds is owned by the outer, embedding application. For example, a malicious ad embedded in `publisher.com` might embed an ACG for geolocation access. If the ad's UI mimics that of the publisher, the user may be tricked into thinking that the ACG will grant access to the publisher rather than to the ad.

To prevent such confusion, we enforce the restriction that only a "top-level" or outermost application can embed ACGs by default. We postulate that users are more cognizant of top-

Algorithm 1 : Can Principal X embed ACG G? If `CanEmbed` returns true, the kernel embeds an active instance of ACG G. Otherwise, it embeds an inactive ACG. X's parent may specify whether X may embed a particular ACG either statically when embedding X, or dynamically in response to a kernel-initiated upcall to `CheckNestedPermission()`.

```
1: function CanEmbed($X$, $G$)
2:     if $X$ is a top-level application then
3:         Return true
4:     if $X$.Parent allows $X$ to embed $G$ then
5:         Return CanEmbed($X$.Parent, $G$)
6:     else
7:         Return false
```

level applications than nested ones. However, we allow an application to permit its nested applications to embed ACGs. For this purpose, we introduce `PermissionToEmbedACG`, a new type of (resource-specific) permission. Consider an application A that embeds application X, which attempts to embed an ACG for a resource. Algorithm 1 shows how the kernel determines whether application X may embed an active ACG for that resource. If this permission is denied, the kernel embeds an inactive gadget. If application X already has access to a resource (via prior session or permanent access grants), it retains this access, but embedding X does not grant application A access to that resource — nor do user actions on ACGs in embedded applications grant permissions to their embedding ancestors.

3.4.2 Access Semantics

After capturing a user's intent to grant access — via an ACG or a permission-granting input sequence (Section 3.4.5) — a resource monitor must act on this intent. We discuss the mechanisms for granting access in our implementation in Section 3.5; here we consider the semantics of different types of access.

Figure 3.6 illustrates access durations, their relationship with user actions, and their respective security implications. There are four types of access durations that may be granted to applications: *one-time* (such as taking a picture), *session* (such as video recording), *scheduled* (access scheduled by events, such as sending a monthly SMS reminder for paying rent), and *permanent.*

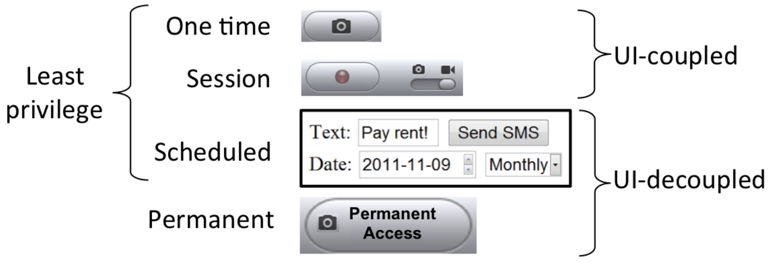


Figure 3.6: **Access Semantics.** Access duration, relationship with user actions, and security implications.

For both one-time and session access, user actions naturally convey the needed access duration. For example, when a user takes a photo, the click on the camera ACG is coupled with the need for one-time access to the camera. Similarly, when recording a video, the user's toggling of the record button corresponds to the need to access the video stream. We characterize this relationship with user actions as *UI-coupled* access. With ACGs, UI-coupled access can be made *least-privilege*. For long-running applications, the system may alert the user about an ongoing session access after the user has not interacted with the application for some time.

In contrast, scheduled and permanent access are *UI-decoupled*: the user's permission-granting actions are decoupled from the access. For example, the user might grant an application permanent camera access; later accesses to this resource need not be directly coupled with any user actions. It is more difficult to make UI-decoupled access least-privilege.

For scheduled scenarios, we propose *schedule ACGs* — using common triggers like time or location as scheduling parameters — to achieve least-privilege access. For example, the user explicitly schedules access for certain times (e.g., automatically access content every 12 hours to back it up) or when in a certain location (e.g., send an SMS when the location criteria is met). In these scenarios, users must already configure these conditions within the application; these configurations could be done via a schedule ACG presented together with resource ACGs in the context of the application.

However, scheduled access does not encompass unconventional events (e.g., when a game's high score is topped). For those scenarios, applications must request permanent access. In our system, permanent access configuration is done via an ACG. To encourage developers to avoid requesting unnecessary permanent access, RMs can require additional user interactions (e.g., confirmation prompts), and security professionals or app store reviewers can apply greater scrutiny (and longer review times) to such applications. Our study of popular Android applications (Section 3.6.3) shows that a mere 5% require permanent access to user-owned resources, meaning users' exposure to prompts will be limited and that extra scrutiny is practical.

It is the RM's responsibility to provide ACGs for different access semantics (if at all — devices like the printer and the clipboard may support only one-time access). Each RM is further responsible for configuring appropriate policies for applications running in the background. For example, unlike access to the speaker, background video recording might not be permitted.

The user must be able to revoke access when the associated ACG is not present. A typical approach is to allow the user to issue a secure attention sequence [80] like Ctrl-Alt-Del (or to navigate system UI) to open a control panel that shows the access control state of the system and allows revocation. Prior work also suggests mechanisms for intuitively conveying access status [68].

3.4.3 ACG Composition

Some applications may need to access multiple user-owned resources at once. For example, an application may take a photo and tag it with the current location, requiring both camera and geolocation access, or support video chat, requiring camera and microphone access. It would be burdensome for users to interact with each ACG in turn to grant these permissions. To enable developers to minimize the number of user actions in such scenarios, we allow applications to embed a *composition ACG* (C-ACG) exposed by a *composition resource monitor*. A single user action on a composition ACG grants permission to all involved user-owned resources.

A composition RM serves as a UI proxy for the RMs of the composed resources: when a user interacts with a C-ACG, the C-ACG invokes privileged API calls to the involved resource RMs (available only to the composition RM). The invocation parameters include (1) the application's anonymous ID (informing the RM to which application to grant permissions), (2) an anonymous handle to the application's running instance (to allow return data, if any, to be delivered), and (3) a transaction ID (allowing the application instance to group data returned from different RMs). This design allows the C-ACG to be least-privilege, with no access to the composed resources.

We disallow arbitrary composition, which could allow applications to undermine our goal of least-privilege access. For example, an application that only needs camera access could embed a C-ACG composing all user-owned resources. Instead, the composition RM exposes a fixed set of compositions that can only be extended by system or RM updates. As with other RMs, the composition RM is responsible for the UI of its ACGs. In our evaluation (Section 3.6.7), we find that composition is rarely used today (only three types of composition appear and are used in only 5% of the applications we surveyed), suggesting that this fixed set of compositions would suffice.

3.4.4 Generalization: Application-Specific ACGs

So far, we have presented ACGs for system resources. Now, we generalize this notion to application-specific resources with *application-specific ACGs.*

Today, applications expose APIs allowing other applications to access their services and resources. We argue that an application should also be able to expose ACGs for its user-sensitive services, thereby requiring authentic user actions to permit the corresponding API access. For example, Facebook might expose an ACG for accessing the Facebook friends list. As another example, applications can expose file-picking ACGs, ensuring that only content authentically picked by the user is shared with other applications (see Section 3.5.3).

An application exposing application-specific ACGs must register itself as a RM and specify its ACGs to the kernel.

Embedding an application-specific ACG is similar to a web mashup where `app1.com` embeds an iframe sourced from `app2.com`. One key distinction is that an application-specific ACG grants permissions to its embedding application X only if X's ancestors granted `PermissionToEmbedACG` down the nesting hierarchy (Section 3.4.1).

3.4.5 Permission-Granting Input Sequences

Instead of interacting with UI elements, some users prefer alternate input methods, e.g., using keyboard, mouse, touch, or voice commands. For example, some users prefer keying Ctrl-C to clicking on the "copy" button or prefer drag-and-drop with a mouse to clicking on copy-and-paste buttons. In addition to the visible ACGs exposed by RMs, we thus support permission-granting sequences detected by the kernel. While prior work (Section 3.7) has suggested specific reserved gestures [51, 142, 147], we establish input sequences as first-class primitives for supporting user-driven access control.

System RMs may register permission-granting input sequences with the kernel. By monitoring the raw stream of input from the user, the kernel can detect these authentic input events and dispatch them — along with information about the application with which the user is interacting — to the appropriate RM. The RM then grants the application the associated permission. For example, the camera RM registers the voice command "Take a picture" as a permission-granting input sequence. When the user says this while using an application, the kernel interprets it as a registered input sequence and sends the event and the application information to the camera RM. The camera RM then acts on this user intent to grant the application the permission to take a picture.

Sequence ownership. As with ACG ownership as described in Section 3.4.1, determining to which application a sequence should be applied is more complex with nested applications. We take the same approach here: by default, the sequence is applied to the top-level application, which can permit nested applications to receive permission-granting sequences with the `PermissionToReceiveSequence` permission.

Sequence conflicts. We consider two kinds of sequence conflicts: (1) Two RMs may attempt to define the same global sequence, and (2) applications may assign global sequences to their own application-specific functionality (e.g., using Ctrl-C to abort a command in a terminal). The former must be resolved by the OS vendor in the process of endorsing new

Type	Call Name	Description
syscall	`InitContentTransfer(push or pull, dest or src)`	Triggers the kernel to push or pull content from a principal.
upcall	`StartUp(gadgetId, appId, appHandle)`	Notifies monitor of a new embedded ACG.
upcall	`InputEvent(gadgetId or inputSequence, appId, appHandle)`	Notifies monitor of ACG input event or a recognized sequence.
upcall	`LostFocus(gadgetId, appId, appHandle)`	Notifies monitor when an ACG loses focus.
upcall	`EmbeddingAppExit(gadgetId, appId, appHandle)`	Notifies monitor when application embedding an ACG exits.

Table 3.1: **System Calls and Upcalls for Resource Monitors.** This table shows the system calls available to resource monitors and the upcalls which they must handle. Each upcall is associated with an ACG instance or permission-granting input sequence.

RMs. Note that application-specific RMs are not permitted to register sequences, as it would make the resolution of cross-RM conflicts unscalable. For (2), applications are encouraged to resolve the conflict by using different sequences. However, they may continue to apply their own interpretations to input events they receive and simply ignore the associated permissions. Applications can also implement additional sequences internally; e.g., Vi could still use "yy" for internal copying. However, the data copied would not be placed on the system clipboard unless the user employed the appropriate permission-granting input sequence or corresponding ACG.

3.5 Implementation

We build our new mechanisms into ServiceOS [173], a prototype system that provides the properties described in Section 3.3.1. In particular, the kernel isolates applications according to the same-origin policy, and it controls and isolates the display. Applications have no default access to user-owned resources and may arbitrarily embed each other; embedded applications are isolated. The system supports a browser runtime for executing web applications, as well as some ported desktop applications, such as Microsoft Word.

We extended ServiceOS with about 2500 lines of C# code to allow (1) the kernel to capture user intent via ACGs or permission-granting input sequences (Section 3.5.1), and (2) the appropriate RM to act on that intent (Section 3.5.2). Tables 3.1 and 3.2 summarize

Type	Call Name	Description
syscall	`EmbedACG(location, resource, type, duration)`	Embeds an ACG in the calling application's UI.
upcall	`PullContent(windowId, eventName, eventArgs)`	Pulls content from a principal based on user intent.
upcall	`PushContent(windowId, eventName, eventArgs)`	Pushes content to a principal based on user intent.
upcall	`IntermediateEvent(windowId, eventName, eventArgs)`	Issues a DragEnter, DragOver, or DragLeave to a principal.
upcall	`IsDraggable(windowId, x, y)`	Determines if object under cursor is draggable.
upcall	`CheckNestedPermission(windowId, nestedApp, acgType)`	Determines if nested application may embed ACG.

Table 3.2: **System Calls and Upcalls for Applications.** This table shows the system calls available to applications and the upcalls which they must handle. The `windowId` allows a multi-window application to determine which window should respond to the upcall.

the system calls and application upcalls we implemented to support user-driven access control. Only the kernel may issue upcalls to applications. Section 3.5.3 describes end-to-end scenarios via representative ACGs.

3.5.1 Capturing User Intent

Supporting ACGs

In our system, resource monitors are implemented as separate applications that provide access-control and UI logic and expose a set of access control gadgets.

The `EmbedACG()` system call allows other applications to embed ACGs. These applications must specify where, within the application's portion of the display, the gadget should be displayed, the desired user-owned resource, and optionally the type of gadget (including limited customization parameters) and the duration of access, depending on the set of gadgets exposed by the RM. For instance, to allow a user to take a picture, an application makes an `EmbedACG(`(x, y)`, "camera", "take-picture")` call in its UI code, where "camera" is the resource and "take-picture" is the gadget type (as opposed to, e.g., "configure-settings" or "take-video"). When a user clicks on an ACG, the appropriate RM is notified with an `InputEvent()` upcall that includes the application's ID and a handle for the running instance (both anonymous).

The kernel starts the appropriate RM process (if necessary), binds these embedded regions to that RM, and notifies it of a new ACG with the `StartUp()` upcall.

Supporting Permission-Granting Input Sequences

We focus our implementation on the most commonly used permission-granting input sequences in desktop systems: keyboard shortcuts for the clipboard (via cut, copy, paste) and mouse gestures for the transient clipboard (via drag-and-drop). These implementation details represent the changes required for any sequence (e.g., Ctrl-P, or voice commands).

We modified the kernel to route mouse and keyboard events to our sequence-detection logic. Once a permission-granting sequence is detected, the kernel passes the recognized event (and information about the receiving application) to the appropriate RM with the `InputEvent()` upcall.

Our clipboard RM registers common keyboard shortcuts for cut, copy, and paste (Ctrl-X, Ctrl-C, and Ctrl-V). To implement drag-and-drop, the kernel identifies a drag as a MouseDown followed by a MouseMove event on a draggable object (determined by `IsDraggable()` upcalls to the object's owner), and it identifies the subsequent MouseUp event as a drop. To support existing visual feedback idioms for drag-and-drop, our implementation dispatches `IntermediateEvent()` upcalls to applications during a drag action.

3.5.2 Acting on User Intent

When an RM receives an `InputEvent()` upcall (i.e., user intent is captured), it grants access in one of two ways, depending on the duration of the granted access.

Session or permanent access: set access control state. When an application is granted session or permanent access to a resource, the resource monitor stores this permission in an access control list (ACL). This permission is revoked either directly by the user, or (in the session-based case) when the application exits and the RM receives an `EmbeddingAppExit()` upcall. While the permission remains valid (i.e., is reflected in the

ACL), the RM allows the application to invoke its resource access APIs.

One-time access: discrete content transfer. When an application is granted one-time access, however, it is not granted access to the resource's API directly. Rather, the RM mediates the transfer of one content item from the source to the destination. This design ensures that the user's one-time intent is accurately captured. For example, when an application receives one-time camera access, it should not be able to defer its access until later, when the camera may no longer be directed at the intended scene. To this end, the RM immediately issues an `InitContentTransfer()` system call, prompting the kernel to issue a `PullContent()` upcall to an application writing to the resource (e.g., a document to print), or a `PushContent()` call to an application reading from the resource (e.g., a picture the RM obtained from the camera).

3.5.3 End-to-End: Representative ACGs

We implemented ACGs for a representative set of user-driven access control scenarios, covering device access, cross-application data sharing, and private data access.

One-time ACGs. We implemented a camera RM that exposes a one-time ACG allowing the user to take a photo and immediately share it with the application embedding the ACG. To capture the photo, the RM leverages existing system camera APIs. When a user clicks on the gadget, the RM (1) issues a call to the camera to take a photo, and (2) issues an `InitContentTransfer()` call to the kernel to transfer the photo to the receiving application. Similarly, our clipboard RM exposes one-time ACGs for copy, cut, and paste.

The database of a user's form autocomplete entries (e.g., his or her name, address, and credit card numbers) often contains private data, so we consider it a user-owned resource. We implemented an autocomplete RM that exposes one-time autocomplete ACGs — text boxes of various types, such as last-name, credit-card, and social-security-number. When the user finishes interacting with an autocomplete ACG (entering text, selecting from a drop-down menu, or leaving it blank) and leaves the ACG window, the autocomplete RM receives a `LostFocus()` upcall. It uses this as a signal to update the autocomplete database (if appropriate) and to pass this text on to the embedding application with an

`InitContentTransfer()` call.

Finally, we implemented a composition ACG, combining one-time camera and one-time geolocation access.

Session and permanent ACGs. We implemented a session-based geolocation ACG, allowing the user to start and stop access (e.g., when using a maps application for navigation). When the user starts a new session, the geolocation RM updates its access control state to allow the application embedding the ACG to access geolocation APIs directly. The application can make `GetCurrentLocation()` calls via the RM until the user ends the session (or closes the application). These mechanisms also support permanent access.

Permission-granting input sequences. As described, we implemented permission-granting input sequences for copy-and-paste and for drag-and-drop. To add support for drag-and-drop, we extended the kernel to make the appropriate `IntermediateEvent()` upcalls. Since drag-and-drop is fundamentally only a gesture, the drag-and-drop RM exposes no gadgets; it merely mediates access to the transient clipboard.

ACG-embedding and application-specific ACGs. As a more complex example, we implemented a content-picking ACG (a generalization of "file picker"), which an application can embed to load in the user's data items from other applications. We have two security goals in designing this ACG:

1. Least-privilege access to user content: *Only* the requesting application receives access to *only* the data item picked by the user. The content-picking RM has no access to the picked item.
2. Minimal attack surface: Since the content picker must retrieve content listings from each content provider application, we aim to minimize the attack surface in interactions between the RM and content providers.

Figure 3.7 illustrates our design for content picking. Each content provider exposes an application-specific ACG (Section 3.4.4) called a content-view ACG, which allows least-privilege, user-driven access to user-picked data items in that content provider. Such a content-view ACG allows each content provider to intelligently control user interactions

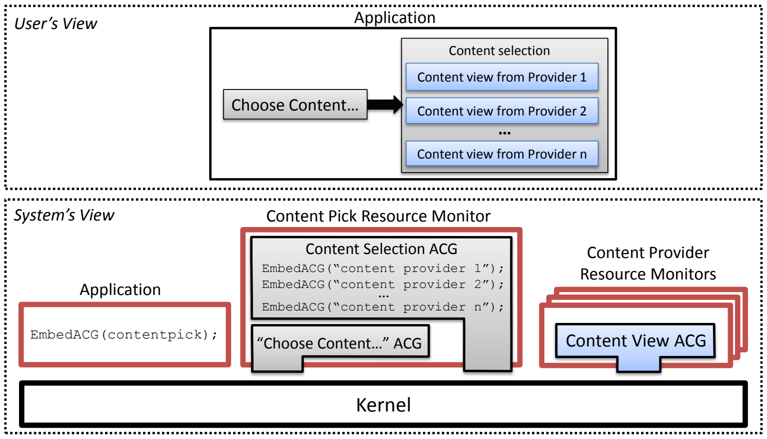

User's View
Application
Content selection
Content view from Provider 1
Choose Content...
Content view from Provider 2
...
Content view from Provider n
System's View
Content Pick Resource Monitor
Content Selection ACG
EmbedACG("content provider 1");
EmbedACG("content provider 2");
...
EmbedACG("content provider n");
Application
EmbedACG(contentpick);
"Choose Content..." ACG
Content Provider Resource Monitors
Content View ACG
Kernel

Figure 3.7: **Content Picking via ACGs.** An application may embed a content-picking ACG (the "Choose content..." button above). When the user clicks on this button, the content-picking RM presents a content-selection ACG, which displays available content by embedding application-specific ACGs from each content provider. The user may select content from one of these content-view ACGs, thereby enabling the original embedding application to access it.

with the content based on its semantics (e.g., Foursquare check-ins or Facebook friends are specialized types of content that would lose richness if handled generically).

The content-picking RM exposes two ACGs, a content-picking ACG and a content-selection ACG. The content-picking ACG is a button that opens the content-selection ACG in response to a user click. The content-selection ACG embeds content providers' content-view ACGs. By isolating content-view ACGs from the content-picking RM/ACGs, we minimize the attack surface of the content-picking RM.

The content-selection ACG is special in that it contains other ACGs (the content-view ACGs from content providers). Thus, when a user interacts with a content-view ACG, its content provider grants access to the application embedding the content-picking ACG, but *not* to the content-picking RM. More concretely, when the user selects a data item from a content-view ACG, the kernel issues a `PullContent()` upcall to the content provider and a `PushContent()` upcall to the application embedding the content-picking ACG.

We implemented a content-picking RM that supports both the “open” and “save” functionality, with the latter reversing the destinations of `PullContent()` and `PushContent()` upcalls. To evaluate the feasibility of using this RM with real-world content providers, we implemented an application-specific content-view ACG for Dropbox, a cloud storage application. We leveraged Dropbox’s public REST APIs to build an ACG that displays the user’s Dropbox files using a .NET TreeView form; it implements `PushContent()` and `PullContent()` upcalls as uploads to and downloads from Dropbox, respectively.

Rather than browsing for content, users may wish to search for a specific item. We extended the content-picking RM to also support search by letting users type a search query in the content-selection ACG and then sending the query to each content provider. Content providers respond with handles to content-view ACGs containing results, along with relevance scores that allow the content-selection ACG to properly interleave the results. Note that we only implemented the access-control actions, not the algorithm to appropriately rank the results. Existing desktop search algorithms would be suitable here, though modifications may be necessary to deal with applications that return bad rankings.

3.6 Evaluation

Our evaluation shows that (1) illegitimate access to user-owned resources is a real problem today and will likely become a dominant source of future vulnerabilities. User-driven access control eliminates most such known vulnerabilities — 82% for Chrome, 96% for Firefox (Section 3.6.1). (2) Unlike existing models, our least-privilege model matches users’ expectations (Section 3.6.2). (3) A survey of top Android applications shows that user-driven access control offers least-privilege access to user-owned resources for 95% of applications, significantly reducing privacy risks (Section 3.6.3). Indeed, (4) attackers have only very limited ways of gaining unauthorized access in our system (Section 3.6.4). We find that (5) system developers can easily add new resources (Section 3.6.5), that (6) application developers can easily incorporate user-driven access control (Section 3.6.6), and that (7) our design does not unreasonably restrict customization (Section 3.6.7). (8) Finally, the performance impact is negligible (Section 3.6.8).

		% We Eliminate by Design	
Vulnerability Class	**Example**	**Chrome Bugs**	**Firefox Bugs**
User data leakage	getData() can retrieve fully qualified path during a file drag.	90% (19 of 21)	100% (18 of 18)
Local resource DoS	Website can download unlimited content to user's file system.	100% (10 of 10)	100% (1 of 1)
Clickjacking	Security-relevant prompts exploitable via timing attacks.	100% (4 of 4)	100% (1 of 1)
User spoofing	Application-initiated forced mouse drag.	100% (3 of 3)	100% (4 of 4)
Cross-app exploits	Script tags included in copied and pasted content.	0% (0 of 6)	50% (1 of 2)
	Total	82% (36 of 44)	96% (25 of 26)

Table 3.3: **Relevant Browser Vulnerabilities.** We categorize Chrome [26] and Firefox [120] vulnerabilities that affect user-owned resources; we show the percentage of vulnerabilities that user-driven access control eliminates by design.

3.6.1 Known Access Control Vulnerabilities in Browsers

We assembled a list of 631 publicly-known security vulnerabilities in recent versions of Chrome (2008-2011) [26] and Firefox (v. 2.0-3.6) [120]. We classify these vulnerabilities and find that memory errors, input validation errors, same-origin-policy violations, and other sandbox bypasses account for 61% of vulnerabilities in Chrome and 71% in Firefox. Previous work on isolating web site principals (e.g., [61, 172]) targets this class of vulnerabilities.

Our focus is on the remaining vulnerabilities, which represent vulnerabilities that cross-principal isolation will not address. Of those remaining, the dominant category (30% in Chrome, 35% in Firefox) pertains to access control for user-owned resources. We subcategorize these remaining vulnerabilities and analyze which can be eliminated by user-driven access control. Table 3.3 summarizes our results.

User data leakage. This class of vulnerability either leads to unauthorized access to locally stored user data (e.g., unrestricted file system privileges) or leakage of a user's data across web applications (e.g., focus stealing to misdirect sensitive input). User-driven access control only grants access based on genuine user interaction with ACGs or permission-granting input sequences.

Nine of the 21 vulnerabilities in Chrome are related to autocomplete functionality. The two that we do not address by design are errors that could be duplicated in an autocomplete RM's implementation (e.g., autocompleting credit card information on a non-HTTPS page).

Local resource DoS. These vulnerabilities allow malicious applications to perform denial-of-service attacks on a user's resources, e.g., downloading content without the user's consent to saturate the disk. With user-driven access control, such a download can proceed only following genuine user interaction with ACGs or permission-granting input sequences.

Clickjacking. Both display- and timing-based clickjacking attacks are eliminated by the fact that each ACG's UI stack is completely controlled by the kernel and the RM (Section 3.4.1). In particular, our system disallows transparent ACGs and enforces a delay on the activation of ACGs when they appear.

User spoofing. A fundamental property of user-driven access control is that user actions granting access cannot be spoofed. This property eliminates user spoofing vulnerabilities in which malicious applications can gain access by, e.g., issuing clicks or drag-and-drop actions programmatically.

Cross-application exploits. In cross-application exploits, an attacker uses content transferred by the user to attack another application (e.g., copying and pasting into vulnerable applications allows cross-site scripting). As per our threat model (Section 3.3.3), we consider the hardening of applications to malicious input to be an orthogonal problem. However, we do eliminate one such issue in Firefox: it involves a malicious search plugin executing JavaScript in the context of the current page. This attack is eliminated in our system because cross-application search is possible only via the search RM, which is isolated from its embedding application.

3.6.2 User Expectations of Access Semantics

We conducted two surveys to understand user expectations of access semantics. These studies adhered to human subjects research requirements of our institution.

As part of a preliminary online user survey with 139 participants (111 male and 28 female, ages 18-72), we asked users about their expectations regarding clipboard access. We found that most users believe that copy-and-paste already has the security properties that we enable with user-driven access control. In particular, over 40% (a plurality, $p < 0.0001$) of users believe, wrongly, that applications can only access the global clipboard when the user pastes. User-driven access control would match this expectation.

Following these preliminary results, we designed a second online user survey to assess user expectations regarding accesses to a broader set of resources and a variety of access durations. We administered this survey to 186 users (154 male and 31 female, aged 18 to over 70). Unless otherwise noted, all of the results reported in this section are statistically significant according to Pearson's chi-squared test ($p < 0.05$).

The survey consisted of a number of scenarios in which applications access user-owned resources, accompanied by screenshots from applications. We examined location access, camera access, and the ability to send SMS across one-time, session, and permanent durations. In each scenario, we asked questions to determine (1) when users believe the applications *can* access the resource in question, and (2) when users believe the application *should* be able to access it.

We used screenshots from existing Android applications for better ecological validity — using these instead of artificial applications created for the purposes of this study allows us to evaluate real user expectations when interacting with real applications. 171 (92%) of our participants reported having used a smartphone before, 96 (52%) having used Android.

One-time and session access. For one-time and session access to resources, we asked users about whether or not applications can/should access resources before and after interactions with related UI elements. For example, we asked users whether a map application can/should access their location *before* they click the "current location" button; we then asked how long *after* that button press the application can/should access their location. Figure 3.8 summarizes the results for before the button press, where we find that, on average across all resources, most users believe that applications cannot — and certainly should not be able to — access the resource before they interact with the relevant UI. These trends are similar for after the button press. Note that participants were less sensitive about location access, a trend we address below.

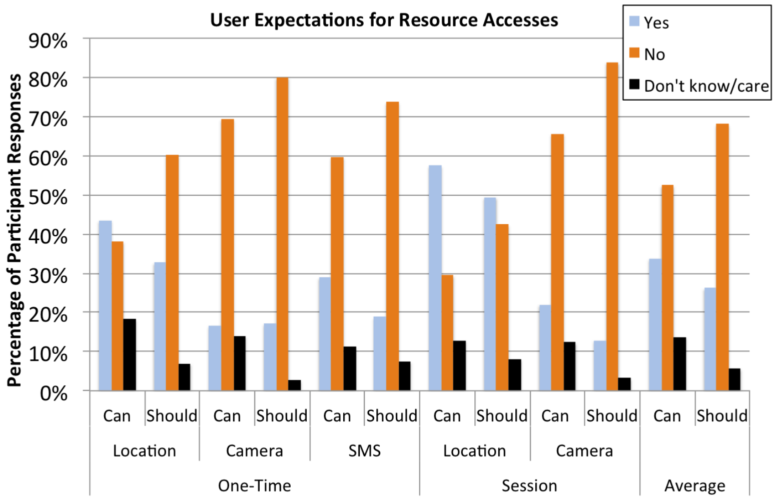


Figure 3.8: **User Expectations of One-Time and Session Access.** For one-time (location, camera, SMS) and session-based (location, camera) access scenarios, participants were asked (1) *can* the application access the resource before they press the button, and (2) *should* it be able to access the resource before they press the button.

To address possible ordering effects, we randomly permuted the order in which we asked about the various resources, as well as about the various durations. We found that question order had no statistically significant effect on responses, except in the case of SMS access. There, we found that participants were statistically significantly more sensitive about an application's ability to send SMS if they were first asked about the camera and/or location.

Despite ordering effects, we find that users are less concerned about applications accessing their location than about accesses to the camera or sending SMS. More formally, we performed a Mixed Model analysis, modeling *ParticipantID* as a random effect. We tested for an interaction between *Order* and *Resource*, finding no effect $(F(4,4) = 1.5435, p = 0.1901)$. We did find *Resource* to be significant $(F(2,2) = 9.8419, p < 0.0001)$, leading us to investigate pairwise differences, where we find lower sensitivity to location access. Indeed, in both one-time and session-based scenarios, most users indicated that applications can access their location uncorrelated to UI interactions — the opposite of their expectations in the camera and SMS scenarios. While most users do believe that these location accesses *should* be tied to UI interactions, this concern was still statistically significantly less than for the camera or SMS. We hypothesize that users may better understand (or consider riskier) the consequences of unwanted access to camera or SMS.

Permanent access. To evaluate users' expectations of permanent access, we presented users with two scenarios: one in which the application pops up a reminder whenever the user is in a preconfigured location, and one in which the application sends an SMS in response to a missed call. 69% of users identified the first application as requiring permanent location access; nevertheless, only 59% of users believed that it should have this degree of access. In the case of the SMS-sending application, 62% of users (incorrectly) believe that the application is only permitted to send SMS when the configured condition is met (a call is missed); 74% of users believe this should be the case. This finding supports our design of a schedule ACG as described in Section 3.4.2.

These results allow us to conclude that user-driven access control — unlike existing systems — provides an access model that largely matches both users' expectations and their preferences regarding how their in-application intent maps to applications' ability to access resources.

3.6.3 Access Semantics in Existing Applications

Using 100 popular Android applications (the top 50 paid and the top 50 free applications as reported by the Android market on November 3, 2011), we evaluated how often applications need permanent access to a resource to perform their intended functionality. We focused on the camera, location (GPS and network-based), the microphone, the phone's calling capability, SMS (sending, receiving, and reading), the contact list, and the calendar. We examined accesses by navigating each application's menu until we found functionality related to each resource requested in its manifest.

Though Android grants an application permanent access to any resource specified in its manifest, we find that most resource accesses (62% of 143 accesses observed across 100 applications) require only one-time or session access in direct response to a user action (UI-coupled). Table 3.4-A summarizes these results. By using an ACG to guide these

	Access Semantics (A)				
	UI-Decoupled		*UI-Coupled*	*Unrelated to primary*	
	(Permanent)	*(Scheduled)*	*(One-Time/Session)*	*functionality (e.g., ads)*	*Unknown*
Camera	0	0	7	0	2
Location (GPS)	1	3	13	3	1
Location (Network)	0	3	9	8	2
Microphone	0	0	14	0	0
Send SMS	0	0	3	0	2
Receive SMS	5	0	0	0	0
Read SMS	2	2	3	0	2
Make Calls	0	0	8	0	2
Intercept Calls	1	0	0	0	0
Read Contacts	2	2	18	0	
Write Contacts	0	0	13	0	2
Read Calendar	1	1	0	0	1
Write Calendar	0	0	1	0	3
Total	12 (8%)	11 (8%)	89 (62%)	11 (8%)	20 (14%)

	Customization (B)	
	Limited	*Arbitrary*
Camera	4	1
Location (GPS) Location (Network)	5	5
Microphone	6	4
Send SMS	3	0
Receive SMS	N/A	N/A
Read SMS	0	2
Make Calls	8	0
Intercept Calls	N/A	N/A
Read Contacts Write Contacts	14	2
Read Calendar	0	0
Write Calendar	0	1
Total	40 (73%)	15 (27%)

Table 3.4: **Analysis of Existing Android Applications.** We analyzed 100 Android applications (top 50 paid and top 50 free), encompassing 143 individual resource accesses. In Table 3.4-A, we evaluate the access semantics required by applications for their intended functionality. We find that most accesses are directly triggered by user actions (UI-coupled). Table 3.4-B examines the customization of existing UIs related to resource access. Entries marked N/A indicate resources for which access is fundamentally triggered by something other than a user action (e.g., incoming SMS). We find that limited customization suffices for most existing applications. Note that some accesses could be handled via a UI-coupled action, but do not currently display such a UI. Thus, the customization columns do not always sum to the UI-coupled column.

interactions, applications would achieve least-privilege access. Indeed, 91 of the applications we examined require only UI-coupled access to all user-owned resources. These results indicate that user-driven access control would greatly reduce applications' ability to access resources compared to the manifest (permanent access) model currently used by Android.

Only nine applications legitimately require UI-decoupled access to at least one resource to provide their intended functionality. We describe these scenarios here and present techniques to reduce the need for permanent access in some cases:

- *Scheduled Access.* Some applications access resources repeatedly, e.g., the current location for weather or user content for regular backups. While these accesses (8% of accesses we examined and 48% of all UI-uncoupled accesses) must occur fundamentally without direct user interaction, these applications could use schedule ACGs as described in Section 3.4.2 to nevertheless achieve least-privilege. This technique reduces the number of applications requiring permanent access from nine to six.
- *Display Only.* Some permanent access scenarios simply display content to the user. For example, many home screen replacement applications require permanent access to the calendar or the ability to read SMS because these data items are displayed on the home screen. At the expense of customization, these applications could simply embed relevant applications that display this content (e.g., the calendar). This technique would prevent another application from requiring permanent access.
- *Event-Driven Access.* Certain event-based permissions fundamentally require UI-decoupled permanent access. For example, receiving incoming SMS (e.g., by anti-virus applications or by SMS application replacements) and intercepting outgoing calls (e.g., by applications that switch voice calls to data calls) are fundamentally triggered by events other than user input.

Overall, we find that only five (5%) of the applications we examined truly require full permanent access to user-owned resources. With user-driven access control, the other 95% of these applications would adhere to least-privilege and greatly reduce their exposure to private user data when not needed.

3.6.4 Gaining Unauthorized Access in Our System

In our system, applications can gain unauthorized access by (1) launching social-engineering attacks against users or (2) abusing permanent (non-least-privilege) access.

On the one hand, ACGs and permission-granting input sequences may make it easier — compared to more intrusive security prompts — for malicious developers to design social engineering attacks. For example, a malicious application may use geolocation ACGs as circle buttons in a tic-tac-toe game, or define an application-specific input sequence that coincides with a permission-granting sequence (e.g., "Push Ctrl-V to win the game!").

However, ACGs and permission-granting input sequences are superior to prompts in their usability and least-privilege properties. Studies suggest that prompts are ineffective as security mechanisms, as users learn to ignore and click through them [118]. Furthermore, usability problems with prompts make them undesirable in real systems that involve frequent access control decisions. Indeed, even the resource access prompts in iOS are generally shown only the first time an application attempts to access a resource, sacrificing least-privilege and making these prompts closer to manifests in terms of security properties. Finally, prompts are also vulnerable to social engineering attacks.

Social engineering indicates an *explicit* malicious intent (or criminal action) from the application vendor, who is subject to legal actions upon discovery. This is in contrast to applications that request unnecessary permanent access or that take advantage of non-least-privilege systems (e.g., to leak location to advertisers) whose malicious intent is much less clear.

Recall from Section 3.4.2 that our system mitigates the permanent access problem by encouraging developers to avoid requesting unnecessary permanent access and by allowing users to revoke it. With just 5% of top Android applications requiring permanent access (Section 3.6.3), these techniques seem practical. The remaining 95% of applications can achieve least-privilege with user-driven access control.

	Extensibility (A)		**Ease of Use (B)**	
	Kernel	*RM*	*Browser*	*MS Word*
Autocomplete	N/A	210	35	N/A
Camera Access	N/A	30	20	25
Content Picking	N/A	285	100	15
Copy-and-Paste	N/A	70	70	60
Drag-and-Drop	70	35	200	45
Global Search	N/A	50	10	N/A
Geolocation Access	N/A	85	15	N/A
Composed Camera+Geolocation	N/A	105	20	N/A

Table 3.5: **Evaluation of Extensibility and Ease of Use.** This table shows the approximate lines of C# code added to the system (Table 3.5-A) and to applications (Table 3.5-B).

3.6.5 Extensibility by System Developers

In general, to support a new user-owned resource in the system, a system developer must implement a resource monitor and its associated ACGs. Table 3.5-A shows the development effort required to implement the example ACGs described in Section 3.5.3, measured in lines of C# code. Adding support for a new resource rarely requires kernel changes — only drag-and-drop required minor modifications to the kernel. RM implementations were small and straightforward, demonstrating that our system is easy to extend.

3.6.6 Ease of Use by Application Developers

We found that it was easy to modify two existing applications — the browser runtime and Microsoft Word 2010 — to utilize user-driven access control. Table 3.5-B summarizes the implementation effort required.

For the browser, we used a wrapper around the rendering engine of an existing commercial browser to implement generic support for copy-and-paste, drag-and-drop, camera and geolocation access, content picking, search, and autocomplete. To let web applications easily embed ACGs, we exposed the `EmbedACG()` system call as a new HTML tag. For example, a web application would embed a camera ACG with the tag `<acgadget resource="camera" type="take-picture">`. We converted several upcalls into their corresponding HTML5 events, such as `ondrop` or `ondragover`. Overall, our browser modifications consisted of a reasonable 450 lines of code; the relevant content is thus exposed to the underlying HTML, requiring no additional effort by the web application developer.

We also added support for certain resources into Microsoft Word by writing a small C# add-in. We replaced the default cut, copy, and paste buttons with our ACGs, and implemented the clipboard and drag-and-drop upcalls to insert content into the current document and to send the currently-selected content to the kernel as necessary. We also embed our camera ACG into Word's UI and modified Word to handle camera callbacks by inserting an incoming photo into the current document. Finally, we added our content picking ACGs to Word and hooked them to document saving and opening functionality. Our add-in only required about 145 total lines of C# and demonstrates that even large real-world applications are easy to adapt to take advantage of user-driven access control.

3.6.7 Customization: Security and Developer Impact

We evaluate the danger of arbitrary customization and the effect of limiting customization in today's applications.

Unvetted arbitrary customization. In our design, ACGs offer limited customization options, but application developers may submit an arbitrarily customized version of an ACG to a vetting process (see Section 3.4.1). Other systems, such as the EROS Trusted Window System (EWS) [147], do not require such vetting. In particular, EWS allows an application to designate arbitrary portions of the screen as copy and paste buttons. The application controls the entire display stack, except for the cursor, which is kernel-controlled. This design allows for full application customizability but risks a malicious application using its freedom to deceive the user.

To evaluate the importance of preventing such an attack, we included in the previously-described 139-participant user study (Section 3.6.2) a task in which the cursor was trustworthy but the buttons were not. As suggested in EWS, the cursor, when hovering over a copy or paste button, displayed text indicating the button's function. Most users (61%) reported not noticing the cursor text. Of those that noticed it, only 44% reported noticing attack situations (inconsistencies between the cursor and the button texts). During the

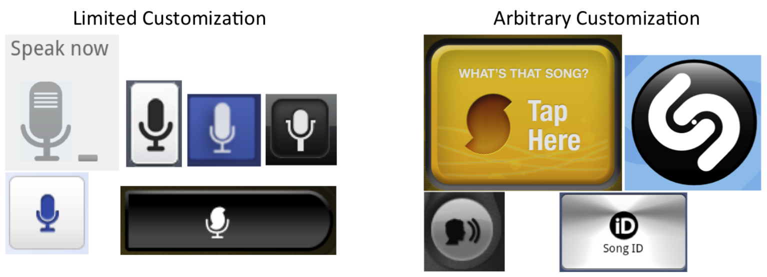


Figure 3.9: **Customization in Android UIs.** Examples of UI elements corresponding to microphone access. The ones on the left could be accommodated with a standard ACG that offers limited customization options (e.g., color). The ones on the right require significant customization, or a change to the application's UI.

trials, users by and large followed the button text, not the cursor, succumbing to 87% of attack trials. Further, introducing the cursor text in the task instructions for about half of the participants did *not* significantly affect either noticing the cursor (Pearson's chi-squared test: $\chi^2(N = 130) = 2.119, p = 0.1455$) or success rate (Mixed Model analysis: $F(1,1) = 0.0063, p = 0.9370$). Thus, it is imperative that the entire UI stack of access-granting features be authentic, as is the case with our ACGs.

Limited customization. To evaluate the sufficiency of limited customization described in Section 3.4.1, we analyzed the 100 Android applications previously described. We examined the degree of customization of UI elements related to accesses to the camera, location (GPS and network-based), the microphone, the phone's calling capability, SMS (sending, receiving, and reading), the contact list, and the calendar.

The results of our evaluation are summarized in Table 3.4-B. In general, we find minimal customization in UI elements related to resource access (e.g., a camera button). In particular, for each UI element we identified a commonly used canonical form, and found that 73% of the UI elements we examined differed only in color or size from this canonical form (using similar icons and text). As an example, Figure 3.9 shows the degree of customization in buttons to access the microphone. We note that applications developers may be willing to replace more than 73% of these UI elements with standardized ACGs; further study is needed to evaluate the willingness of developers to use ACGs in their applications. Nevertheless, we find that the majority of today's UI elements already require only limited customization.

We observed only three types of UI elements associated with multiple resource accesses: microphone with camera in two video chatting applications (e.g., Skype), microphone with location in two applications that identify songs and tag them with the current location (e.g., Shazam), and camera with location in one camera application. These findings support our decision to limit composition (Section 3.4.1).

3.6.8 Performance

We evaluate the performance of user-driven access control, focusing our investigation on drag-and-drop for two reasons: (1) its dataflow is similar or identical to the dataflow of all one-time access-control abstractions and thus its performance will be indicative, and (2) the usability of drag-and-drop (a continuous, synchronous action) is performance-sensitive.

In particular, we evaluate the performance of intermediate drag-and-drop events, which are fired on every mouse move while the user drags an object. We compared our performance to that of Windows/COM for the same events. We ran all measurements on a computer with a dual-proc 3GHz Xeon CPU with 12GB RAM, running 64-bit Windows 7.

In both systems, we measured the time from the registration of the initial mouse event by the kernel to the triggering of the relevant application event handler. The difference was negligible. In Windows, this process took 0.45 ms, averaged over 100 samples (standard deviation 0.11 ms); in our system, it averaged 0.47 ms (standard deviation 0.22 ms).

3.7 Related Work

Philosophically, user-driven access control is consistent with Yee's proposal to align usability and security in the context of capability systems [178]. We presented state-of-the-art permission models on commercial client platforms, such as iOS, Android, and browsers, in Section 3.2. We address other related approaches here.

User action requirements. Several approaches provide in-context permission granting, relying on user actions as permission-granting triggers. These user action requirements are usually simple, such as a click or keystroke, but may extend to reserved shortcuts (such as Ctrl-V for paste). These designs may be seen as predecessors of our work.

For example, Flash introduced a user-initiated action requirement [5], restricting paste to the Ctrl-V shortcut and requiring a click or keystroke for other permissions (such as fullscreen or file system access). Silverlight employs a similar model [114]. While more user friendly than prompts, a simple user action requirement can be easily circumvented by an application that asks (or waits for) the user to perform an unrelated (but permission-granting) action.

Several research systems have also advocated the use of specific reserved gestures for access to the clipboard (similar to Flash's Ctrl-V requirement). For example, the EROS Trusted Window System (EWS) [147], the Qubes OS [142], and Tahoma [31] advocate reserved gestures for copy-and-paste. NitPicker [51] and EWS also support drag-and-drop, and EWS additionally supports copy-and-paste via a user click (on a system-controlled transparent window overlaid on an application's custom copy or paste button).

Finally, CapDesk [116] and Polaris [157] are experimental capability-based desktop systems that apply a user action requirement to file picking. Both give applications minimal privileges, but allow users to grant applications permission to access individual files via a "powerbox" (a trusted system file picker). Recent systems like Mac OS X Lion [8] and Windows 8 [110] have adopted this approach for file picking.

In our work, we generalize these ideas beyond the clipboard and the file system to all sensitives resources, and we design a system that can accurately link a user's action to his or her permission-granting intent.

Inferring user intent. Beyond a simple user action requirement, some researchers have explored systems that directly infer a user's permission-granting intent. For example, Shirley and Evans [149] propose a system (prototyped only for file resources) that tries to infer a user's access control intent from the history of user behavior. BLADE [96] attempts to verify a user's intent to download a file by detecting user interactions with browser download dialogs. In both cases, the systems tries to infer user intent from his or her interactions with an existing dialog; in our work, we design a new type of interaction from which the system can robustly capture user intent without the need for inference.

Visualization and auditing. In addition to — or instead of — other permission-granting mechanisms, some designs allow users to visualize current permission use or audit access after the fact. For example, many cameras (such as on laptops) light up an LED while recording (although users cannot necessarily determine which application is using the camera). Research proposals at application granularity include "sensor-access widgets" [68], user interface elements overlaid on an application that display feedback about the application's current access to the corresponding sensor, and that act as a control point for the user to grant or revoke access. Other work [48] has proposed automatically allowing applications to perform actions that have low severity or are easily reverted (e.g., changing the home screen wallpaper), and coupling this ability with an auditing mechanism that allows users to identify misbehaving applications. Such visualization and auditing techniques can complement the approach that we take in our work.

Mechanism without policy. Finally, a host of prior work has explored access control mechanisms without specifying how the corresponding policies should be captured from user actions. For example, multi-level security systems like SELinux [124] classify information and users into sensitivity levels to support mandatory access control security policies. Information flow control techniques (e.g., [181]) can provide OS-level enforcement of information flow policies, and have been applied in modern contexts such as smartphones (e.g., TaintDroid [42] provides information flow tracking for Android).

Though these mechanisms can help enforce access control policies, they do not address the question of how users should specify these policies. Studies show that access control policies are notoriously difficult to configure correctly, both among system administration experts [32] as well as among ordinary users attempting to manage their own data (e.g., on Facebook [100]). The question of how to allow users to specify such policies in a usable way, without requiring them to constantly understand and manage low-level access control decisions, is the focus of this work.

3.8 Summary

In this chapter, we advocated *user-driven access control* to address the challenge of application permission granting in modern client platforms. Unlike existing models (such as manifests and system prompts) where the system has no insight into the user's behavior in an application, we allow an application to build privileged, permission-granting user actions into its own context. This enables an in-context, non-disruptive, and least-privileged permission system. We introduce *access control gadgets* as privileged, permission-granting UIs exposed by user-owned resources. Together with permission-granting input sequences (such as drag-and-drop), they form the complete set of primitives to enable user-driven access control in modern operating systems.

Our evaluation shows that illegitimate access to user-owned resources is a real problem today and will likely become a dominant source of future vulnerabilities. User-driven access control can potentially eliminate most such vulnerabilities (82% for Chrome, 96% for Firefox). Our user studies indicate that users *expect* least-privilege access to their resources, which the existing models fail to deliver, and which user-driven access control can offer. By studying access semantics of top Android applications, we find that user-driven access control offers least-privilege access to user-owned resources for 95% of the applications, significantly reducing privacy risks. Our implementation experience suggests that it is easy for system developers to add support for new resource types and for application developers to incorporate ACGs into their applications. With these results, we advocate that modern client platforms adopt user-driven access control for scenarios in which explicit user actions are common. In Chapter 5, we develop another permission granting model, world-driven access control, for platforms and applications in which user interactions are sparse.

Chapter 4

SECURING EMBEDDED USER INTERFACES IN MODERN OPERATING SYSTEMS

In Chapter 3, we introduced access control gadgets (ACGs) — system-trusted embedded user interface elements — to allow the operating system to capture a user's implicit intent to grant a permission to an application. For the user's intent to be captured authentically, we discussed several ad-hoc measures to protect ACGs from potentially malicious embedding applications. However, the need to secure embedded user interfaces applies not only to ACGs: today's web and smartphone applications commonly embed third-party user interfaces like advertisements and social media widgets. In this chapter, we thus consider more broadly the security and privacy issues associated with embedded third-party user interfaces, further tackling the first key challenge, embedded third-party principals, identified in Chapter 1. These ideas first appeared in a 2012 paper on security requirements and techniques for user interface toolkits [136] and a 2013 paper applying and refining these techniques in the context of Android, a smartphone operating system [137].

4.1 Overview

Modern web and smartphone applications commonly embed third-party content within their own interfaces. Websites embed iframes containing advertisements, social media widgets (e.g., Facebook's "Like" or Twitter's "tweet" button), Google search results, or maps. Smartphone applications include third-party libraries that display advertisements or provide billing functionality.

Including third-party content comes with potential security implications, both for the embedded content and the host application. For example, a malicious host may attempt to eavesdrop on input intended for embedded content or forge a user's intent to interact with it, either by tricking the user (e.g., by clickjacking) or by programmatically issuing input events. On the other hand, a malicious embedded principal may, for example, attempt to take over a larger display area than expected.

The web security model has evolved over time to address these and other threats. For example, the same-origin policy prevents embedded content from directly accessing or manipulating the parent page, and vice versa. As recently as 2010, browsers added the `sandbox` attribute for iframes [14], allowing websites to prevent embedded content from running scripts or redirecting the top-level page. However, other attacks — like clickjacking — remain a serious concern. Malicious websites frequently mount "likejacking" attacks [154] on the Facebook "Like" button, in which they trick users into sharing the host page on their Facebook profiles. If Facebook suspects a button as part of such an attack, it asks the user to confirm any action in an additional popup dialog [47] — in other words, Facebook falls back on a non-embedded interface to compensate for the insecurity of embedded interfaces.

While numerous research efforts have attempted to close the remaining security gaps with respect to interface embedding on the web [70, 159, 172], they struggle with maintaining backwards compatibility and are burdened with the complexity of the existing web model. By contrast, Android, which to date offers no cross-application embedding, offers a compelling opportunity to redesign secure embedded interfaces from scratch.

Today, applications on Android and other mobile operating systems cannot embed interfaces from another principal; rather, they include third-party libraries that run in the host application's context and provide custom user interface elements (such as advertisements). On the one hand, these libraries can thus abuse the permissions of or otherwise take advantage of their host applications. On the other hand, interface elements provided by these libraries are vulnerable to manipulation by the host application. For example, Android applications can programmatically click on embedded ads in an attempt to increase their advertising revenue [136]. This lack of security also precludes desirable functionality from the Android ecosystem. For example, the social plugins that Facebook provides on the web (e.g., the "Like" button or comments widget) are not available on Android.

Previous research efforts for Android [131, 148] have focused only on one interface embedding scenario: advertising. As a result, these systems, while valuable, do not provide complete or generalizable solutions for interface embedding. For example, to our knowledge, no existing Android-based solution prevents a host application from eavesdropping on input to an embedded interface.

In this chapter, we explore what it takes to support secure embedded UIs, first *in a user interface toolkit architecture in general* (Section 4.4), and then *on Android specifically* (Sections 4.5–4.6). We outline UI-level security threats that motivate our development of a set of desired security properties for embedded user interfaces (Section 4.2), along with the techniques to achieve these properties. We systematically analyze existing systems, including browsers, with respect to whether and how they provide these properties (Section 4.3). Informed by this analysis, we modify the Android framework to support cross-principal interface embedding in a way that meets our security goals. We evaluate our implementation using case studies that rely on embedded interfaces (Section 4.7), including: (1) advertisement libraries that run in a separate process from the embedding application, (2) Facebook social plugins, to date available only on the web, and (3) access control gadgets (introduced in Chapter 3) that allow applications to access sensitive resources (like geolocation) only in response to real user input. Through our implementation experience, we consolidate and evaluate approaches from our own and others' prior work.

Today's system developers wishing to support secure embedded user interfaces have no systematic set of techniques or criteria upon which they can draw. Short of simply adopting the web model by directly extending an existing browser — which may be undesirable for many reasons, including the need to maintain backwards compatibility with the existing web ecosystem and programming model — system developers must (1) reverse-engineer existing techniques used by browsers, and (2) evaluate and integrate research solutions that address remaining issues. In addition to presenting the first secure interface embedding solution for Android, this chapter provides a concrete, comprehensive, and system-independent set of criteria and techniques for supporting secure embedded user interfaces.

4.2 Motivation and Goals

To motivate the need for secure embedded user interfaces, we describe (1) the functionality enabled by embedded applications and interfaces, and (2) the security concerns associated with this embedding. We argue that interface embedding often increases the usability of a particular interaction — embedded content is shown in context, and users can interact with multiple principals in one view — but that security concerns associated with cross-principal UI embedding lead to designs that are more disruptive to the user experience (e.g., prompts).

4.2.1 Functionality

Third-party applications. web and smartphone applications often embed or redirect to user interfaces from other sources. Common use cases include third-party advertisements and social sharing widgets (e.g., Facebook's "Like" button or comment feed, Google's "+1" button, or a Twitter feed). Other examples of embeddable content include search boxes and maps.

On the web, content embedding is done using HTML tags like `iframe` or `object`. On smartphone operating systems like iOS and Android, however, applications cannot directly embed UI from other applications but rather do one of two things: (1) launch another application's full-screen view (via an Android Intent or an iOS RemoteViewController [18]) or (2) include a library that provides embeddable UI elements in the application's own process. The former is generally used for sharing actions (e.g., sending an email) and the latter is generally used for embedded advertisements and billing.

System UI. Security-sensitive actions often elicit system interfaces, usually in the form of prompts. For example, Windows users are shown a User Account Control dialog [112] when an application requires elevation to administrative privileges, and iOS and browser users must respond to a permission dialog when an application attempts to access a sensitive resource (e.g., geolocation).

Because prompts result in a disruptive user experience, the research community has explored using embedded system interfaces to improve the usability of security-sensitive interactions like resource access. In particular, Chapter 3 of this dissertation described access control gadgets (ACGs), embeddable UI elements that — with user interaction — grant applications access to various system resources, including the camera, the clipboard, the GPS, etc. For example, an application might embed a location ACG, which is provided by the system and displays a recognizable location icon; when the user clicks the ACG, the embedding application receives the current GPS coordinates. As we describe below, ACGs cannot be introduced into most of today's systems without significant changes to those systems.

4.2.2 Threat Model and Security Concerns

We consider user interfaces composed of elements from different, potentially mutually distrusting principals (e.g., a host application and an embedded advertisement or an embedded ACG). Host principals may attempt to manipulate interface elements embedded from another principal, and embedded principals may attempt to manipulate those of their host. We assume that the system itself is trustworthy and uncompromised.

We observe that while web and smartphone applications rely heavily on third-party content and services, the associated third-party user interface is not always actually embedded inside of the client application. For example, websites redirect users to PayPal's full-screen page, OAuth authorization dialogs appear in pop-up or redirect windows, and wwebeb users who click on a Facebook "Like" button that is suspected of being part of a clickjacking attack will see an additional pop-up confirmation dialog. We observe two main security-related reasons for the choice not to embed or not to be embedded.

One reason is concern about phishing. If users become accustomed to seeing embedded third-party login or payment forms, they may become desensitized to their existence. Further, because users cannot easily determine the origin (or presence) of embedded content, malicious applications may try to trick users into entering sensitive information into spoofed embedded forms (a form of phishing). Thus, legitimate security-sensitive forms are often shown in their own tab or window.

Our goal in this chapter is *not* to address such phishing attacks, but rather to evaluate and implement methods for securely embedding one legitimate (i.e., not spoofed) application within another. (While extensions of existing approaches, such as SiteKeys, may help mitigate embedded phishing attacks, these approaches do have limitations [144] and are orthogonal to the goals of this chapter.[1])

More importantly — and the subject of this chapter — even legitimate embedded inter-

[1]We also note that phished or spoofed interfaces are little threat if they do not accept private user input — for example, clicking on a fake ACG will not grant any permissions to the embedding application.

faces may be subject to a wide range of attacks, or may present a threat to the application or page that embeds them. In particular, embedded UIs or their parents may be subject to:

- **Display forgery attacks**, in which the parent application modifies the child element (e.g., to display a false payment value), or vice versa.
- **Size manipulation attacks**, in which the parent application violates the child element's size requirements or expectations (e.g., to secretly take photos by hiding the camera preview [161]), or the child element sets it own size inappropriately (e.g., to display a full-screen ad).
- **Input forgery attacks**, in which the parent application delivers forged user input to a child element (e.g., to programmatically click on an advertisement to increase ad revenue), or vice versa.
- **Clickjacking attacks**, in which the parent application forces or tricks the user into clicking on an embedded element [70] using visual tricks (partially obscuring the child element or making it transparent) or via timing-based attacks (popping up the child element just as the user is about to click in a predictable place).
- **Focus stealing attacks**, in which the parent application steals the input focus from an embedded element, capturing input intended for it, or vice versa.
- **Ancestor redirection attacks**, in which a child element redirects an ancestor (e.g., the top-level) application or page to a target of its choice, without user consent.
- **Denial-of-service attacks**, in which the parent application prevents user input from reaching a child element (e.g., to prevent a user from clicking "Cancel" on an authorization dialog), or vice versa.
- **Data privacy attacks**, in which the parent or child extract content displayed in the other.
- **Eavesdropping attacks**, in which the parent application eavesdrops on user input intended for a child element (e.g., a sensitive search query), or vice versa.

4.2.3 Security Goals

Motivated by the above challenge, this chapter proposes, implements, and evaluates the following desired security properties for embedded cross-principal user interfaces:

1. *Display Integrity.* One principal cannot alter the content or appearance of another's interface element, either by direct pixel manipulation or by element size manipulation. This property prevents display forgery and size manipulation attacks.
2. *Input Integrity.* One principal cannot programmatically interact with another's interface element. This property prevents input forgery attacks.
3. *Intent Integrity.* First, an interface element can implement (or request that the system enforce) protection against clickjacking attacks. Second, one principal cannot prevent intended user interactions with another's interface element (denial-of-service). Finally, based on our implementation experience (Section 4.6), we add two additional requirements: an embedded interface element cannot redirect an ancestor's view without user consent, and no interface element can steal focus from another interface element belonging to a different principal.
4. *Data Isolation.* One principal cannot extract content displayed in, nor eavesdrop on user input intended for, another's interface element. This property prevents data privacy and eavesdropping attacks.
5. *UI-to-API Links.* APIs can verify that they were called by a particular principal or interface element.

These properties assume that principals can be reliably distinguished and isolated, either by process separation, run-time validation (e.g., of the same-origin policy), or compile-time validation (e.g., using static analysis).

4.3 Analysis of UI Embedding Systems

To assess the spectrum of solutions and to inform our own design and implementation choices, we now step back and analyze prior web and Android based solutions for cross-application embedded interfaces with respect to the set of security properties described in Section 4.2.3. This analysis is summarized in Table 4.1.

Category	*Security Requirement*	**Browsers**	**Android**	**AdDroid [131]**	**AdSplit [148]**
Display Integrity	Prevents direct modification	✓	✗	✗	✓
	Prevents size manipulation	✗	✗	✗	✗
Input Integrity	Prevents programmatic input	✓	✗	✗	✓
Intent Integrity	Clickjacking protection	✗	✓	✓	✓
	Prevents input denial-of-service	✓	✗	✗	✗
	Prevents focus stealing	✓	✗	✗	✓
	Prevents ancestor redirection	✓	✗	✗	✗
Data Isolation	Prevents access to display	✓	✗	✗	✓
	Prevents input eavesdropping	✓	✗	✗	✗
UI-to-API Links	APIs can verify caller	✓	✗	✗	✓

Table 4.1: **Analysis of Existing Systems.** This table summarizes, to the best of our knowledge, the UI-level security properties (developed in Section 4.2.3) achieved by existing systems. Table 4.2 similarly analyzes our implementation.

4.3.1 Browsers

Browsers support third-party embedding by allowing web pages to include iframes from different domains. Like all pages, iframes are isolated from their parent pages based on the same-origin policy [169], and browsers do not allow pages from different origins to capture or generate input for each other.

However, an iframe's parent has full control of its size, layout, and style, including the ability to make it transparent or overlay it with other content. These capabilities enable clickjacking attacks. While there are various "framebusting" techniques that allow a sensitive page to prevent itself from being framed in an attempt to prevent such attacks, these techniques are not foolproof [143]. More importantly, framebusting is a technique to prevent embedding, not one that supports secure embedding.

Additionally, while an iframe cannot read the URL(s) of its ancestor(s), it can change the top-level URL, redirecting the page without user consent. Newer version of some browsers allow parent pages to protect themselves by using the `sandbox` attribute for iframes; thus, we've indicated that the web prevents such attacks in Table 4.1. However, we observe that it may be desirable to allow user actions to override such a restriction, and we describe how to achieve such a policy in later sections.

Research browsers and browser operating systems (e.g., Gazelle [172] and IBOS [159]) provide similar embedded UI security properties as traditional browsers, and thus we omit them from Table 4.1. Gazelle partially addresses clickjacking by allowing only opaque cross-origin overlays, but this policy is not backwards compatible. Furthermore, malicious parent pages can still obscure embedded content by partially overlaying additional content on top of sensitive iframes. We discuss additional work considering clickjacking in Section 4.10.

4.3.2 Android

Unlike browsers, Android has no notion of isolated iframes; rather, application developers wishing to embed third-party content must do so by including libraries that run in the host application's execution context. As a result, stock Android provides almost no security properties for or from embedded user interfaces: parent and child interface elements running in the same execution context can manipulate each other arbitrarily. We further detail the implications of Android's UI-level security weaknesses through case studies in Section 4.5.

Two recent research efforts [131, 148] propose privilege separation to address security concerns with Android's advertising model (under which third-party ad libraries run in the context of the host application). AdDroid's approach [131] introduces a system advertising service that returns advertisements to the AdDroid userspace library, which displays them in a new user interface element called an AdView. While this approach successfully removes untrusted ad libraries from applications, it does not provide any additional UI-level security properties for the embedded AdView beyond what is provided by stock Android (see Table 4.1). For example, it does not prevent the host application from engaging in clickfraud by programmatically clicking on ads.

AdSplit [148], on the other hand, fully separates advertisements into distinct Android applications (one for each host application). AdSplit achieves the visual embedding of the ad's UI into the application's UI by overlaying the host application, with a transparent region for the ad, on top of the ad application. It prevents programmatic clickfraud attacks by authenticating user input using Quire [35]. As summarized in Table 4.1, AdSplit meets the majority of security requirements for embedded UIs. Indeed, the requirements it meets are sufficient for embedded advertisements. Because it does not meet all of the requirements, however — most importantly, it does not prevent input eavesdropping — AdSplit would not be well-suited as a generalized solution for embedded UIs.

4.4 Security-Aware User Interface Toolkit Architecture

In this section, we first consider securing cross-principal user interfaces from the perspective of a user interface toolkit; in the remainder of this chapter, we then apply the techniques developed here specifically to Android, refining them in the process.

User interface toolkits help to reduce barriers in the design and development of modern graphical interfaces [121, 126]. In aiming to provide maximal flexibility and expressivity, existing toolkit research generally makes an implicit assumption that developers have full control of an interface (e.g., [36, 38, 40, 71, 121]). However, as applications move towards interfaces composed of elements from different principals, this assumption can pose significant security risks, as discussed in Section 4.2. We argue that a user interface toolkit should thus consider security as a first-order goal.

We introduce a user interface toolkit architecture that achieves the security properties described in Section 4.2.3 by (1) isolating interface elements by principal and (2) maintaining specific invariants with respect to the interface layout tree. The toolkit does so while retaining developer flexibility to (3) expose model-level APIs, (4) compose elements, and (5) display feedback across principals. We describe each of these design points in turn.

4.4.1 Achieving Security Properties

Principals and Permissions

Our architecture separates mutually distrusting application components, both interface code and application code, by principal. For example, system code belongs to one principal (referred to as "system-trusted" throughout this section), an application belongs to another principal, and third-party elements belong to their own principals (e.g., an ad library, a PayPal payment button, or a Google login field). Figure 4.1 shows how parts of an application

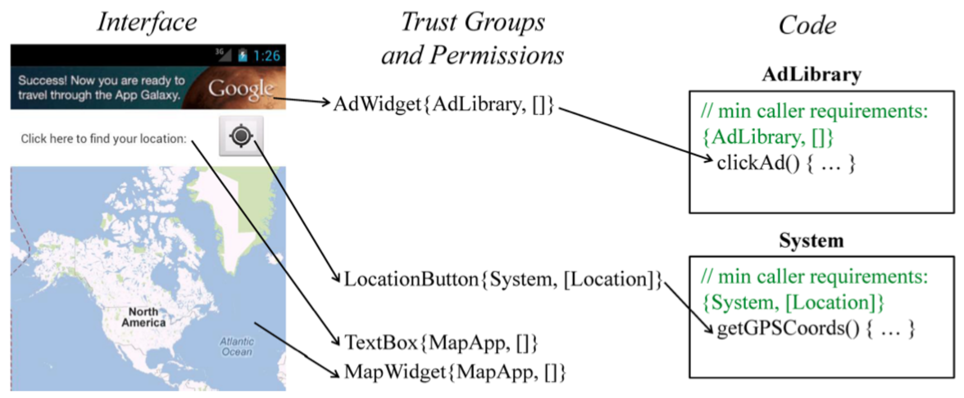

Interface
Trust Groups and Permissions
Code
Success! Now you are ready to travel through the App Galaxy.
Google
Click here to find your location:
North America
Atlantic Ocean
AdWidget{AdLibrary, []}
LocationButton{System, [Location]}
TextBox{MapApp, []}
MapWidget{MapApp, []}
AdLibrary
// min caller requirements: {AdLibrary, []}
clickAd() { … }
System
// min caller requirements: {System, [Location]}
getGPSCoords() { … }

Figure 4.1: **Principals and Permissions.** An example of how an element's principal and permissions map to accessible APIs. An element with principal A and permissions B and C is designated ElementName {A, [B, C]}.

might map to principals; we explain the details of this figure throughout this subsection.

We note that our architecture is agnostic to a system's specific implementation of principal separation. For example, browsers separate principals by dynamic enforcement of the same-origin policy, LayerCake — our modified version of Android that we describe in Section 4.6 — separates principals into distinct Android applications (isolated by running in different processes), and our initial Android prototype (described in our 2012 paper [136]) separated Android UI elements based on Java package names and (hypothesized but not implemented) static analysis.

Principal assignment. In order to isolate elements from different sources, our toolkit must associate all code and all interface elements with a principal. We leave it up to applications, libraries, and the system to associate their own non-interface code with principals at any granularity, but we must consider the appropriate model for assigning interface elements to principals. In particular, our toolkit architecture must support two types of scenarios. In the first, an application embeds a sensitive element exposed by another principal, such as a system-trusted camera button (i.e., a camera access control gadget). In the second, an application simply uses a generic toolkit element (such as a standard button) which should become part of the application's own principal.

To support these scenarios, we introduce two types of interface elements: *fixed-trust-group elements* and *owner-bound elements*. A fixed-trust-group element has a set principal that does not depend upon which entity instantiates or embeds it (e.g., a camera button may belong to the system group). An owner-bound element, such as a standard button, adopts the principal of the code that instantiates it.

Restricting application, system, and library APIs. Our architecture can use principals to create the desired link between interface code and application, system, or library code (UI-to-API Links) by simply denying API access to callers from unauthorized principals.

However, principals may sometimes not be sufficiently granular for an API's access control policy. For instance, the system may wish to restrict the `takePicture()` API only to the system-provided camera button, not to all system interface elements. This policy embodies the standard security principle of least privilege, as arbitrary system components do not need access to the camera. Thus, even if other system elements could be manipulated into invoking the `takePicture()` API, those calls would fail.

To allow such fine-grained access control policies, we associate each interface element with a set of permissions in addition to its principal. Taken together, these allow the system or an application to restrict certain APIs (e.g., `takePicture()`) to code originating from interface elements of an appropriate principal and with sufficient permissions (e.g., an element with principal "system" and a permission list containing the "camera" permission).

In this section, we designate an interface element with principal A and permissions B and C as Element{A, [B, C]}. Figure 4.1 shows an example of how elements map to principals and permissions, which in turn map to accessible APIs in application, system, or library code. In the common case (such as the AdWidget in Figure 4.1), an element's permissions list will be empty, giving it access to only those APIs that do not require any additional permissions.

Restricting interface APIs. Interface elements may need to protect certain methods and expose others. For example, an ad element would not wish to allow `performClick()` to be called by another principal, but it may wish to expose a method to set advertisement keywords. Similarly, an application should be able to attach a callback to a system camera element to receive a photo after it is taken.

Interface elements are thus responsible for defining an access policy for their methods. By default, only code within the same principal and the system's principal can access all of an element's methods. We note that an alternate default would allow all access, relying on developers to selectively restrict sensitive methods. We opt for the stricter default, lowering the threshold to implementing secure interfaces under the assumption that most interfaces do not require crossing security boundaries.

Restrictions on relevant methods provide the following protections, which are subsets of our security properties:

- Code cannot manipulate the display of elements belonging to other principals (Display Integrity). For example, the methods `setBackground()` or `setTransparency()` can be inaccessible to code from another principal.
- Code cannot extract data from elements belonging to other principals (Data Isolation). For example, the method `getText()` can be inaccessible.
- Code cannot programmatically click on elements belonging to other principals (Input Integrity).
- Code cannot enable or disable elements belonging to other principals (Intent Integrity).

Another aspect of Intent Integrity is the ability to prevent clickjacking attacks, in which a malicious parent reveals a sensitive interface element from another principal just as the user is about to click in a predictable location. An element can use system-provided information about its current visibility to implement a clickjacking-protection policy, such as becoming enabled only after being fully visible for some time — as in Chrome (Issue 52868 [26]) and Firefox (Advisory 2008-08 [120]).

Interface Layout Tree Invariants

Using principals and permission lists, our architecture achieves a number of desired security properties. However, applications are still susceptible to security weaknesses stemming from the way in which traditional user interface toolkits organize interface elements into a layout tree (where an element is the parent of any elements it embeds).

- *Insecure layout:* Although principals and permissions restrict method calls on elements, code can still manipulate an element's display using layout techniques (i.e., violate Display Integrity). For instance, an application wishing to hide a required interface element could simply cover that portion of an interface — e.g., the shield icon, required by the Windows User Account Control [138] to adorn any button which will result in actions requiring administrator privileges. Similarly, an application wishing to force a user into clicking on a system-trusted location button could simply make this button much larger than surrounding buttons. In traditional user interface toolkits, parent nodes can control the size and layout of their children. As a result, an application that embeds an element of another principal has complete control over the drawing and layout of the embedded element.
- *Insecure input:* In a traditional layout tree, a malicious application can eavesdrop on, block, or modify events as they trickle down the tree. For example, an application could embed a third-party login element and then steal a user's password by eavesdropping on key events as they propagate down the tree (thus violating Data Isolation). Other attacks could modify the picking code to redirect input to a different interface element (e.g., modifying a legitimate click event to move its location onto a secure button) or prevent an event's propagation entirely. The fundamental challenge is that a single layout tree contains nodes from different principals, thus introducing the potential for untrusted nodes to access events before they reach their intended recipient.

Layout tree invariants. To mitigate these weaknesses, our toolkit departs from traditional toolkits to enforce the following invariants: (1) the root node of an application's layout tree must be a system node, and (2) only system nodes may have children of a different principal. Because we can trust the system itself to provide elements with the layout attributes they request (subject to any necessary conflict resolution), these invariants mitigate the insecure layout weakness. We can also trust the system to provide accurate information about an element's position and visibility (enabling the clickjacking protection described above). Furthermore, only nodes belonging to the same principal or the system's principal will see

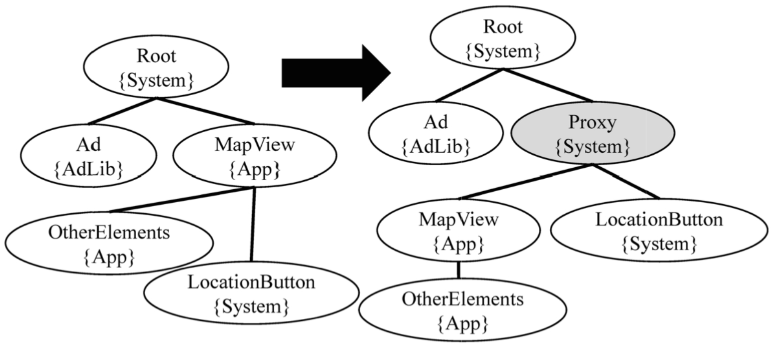


Figure 4.2: **Layout Tree Transformation.** The intended layout tree for the application shown in Figure 4.1 is transformed to include a proxy node, enforcing the invariant that only system-trusted nodes may have children of a different principal.

input events propagating down the tree, thus eliminating the eavesdropping threat.

Conflicts in visual space. Although our invariants dictate that only system nodes may have children from another principal, applications may wish to visually embed elements from different principals. Consider the example in Figure 4.1, where an interface consists of an application map element containing a system-trusted location button. The first invariant is easy to satisfy by placing the entire application in a system-trusted frame. However, the second invariant is more difficult to satisfy while achieving the desired visual effect, as it prevents the map element from being the parent of the location button.

We solve this problem by introducing a system-trusted *proxy node* into the layout tree. Figure 4.2 shows the resulting transformation of the tree. In particular, the proxy node becomes a parent of both the map element and the location button, overlaying them as intended by the application. This transformation preserves the trust hierarchy for layout and for event propagation. We describe in later sections how this transformation can be implemented automatically and transparently to applications, which can continue to manipulate a view of the layout tree that matches the intended visual layout.

A final issue to resolve is layout requests that are conflicting or otherwise cannot be satisfied. The parent element traditionally controls space available to a child element, but this approach may violate security properties (e.g., an application may tell an element to paint itself so small that important context is hidden from a user). On the other hand, a malicious child that controls its own size could draw outside the acceptable bounds (e.g., a malicious ad library that takes over the entire screen).

In the case of a system-trusted child element (e.g., a location button), we rely on our assumption that the system is trustworthy. Our architecture enforces the minimum requested size for system-trusted elements, even if the size allocated by the parent is smaller than this minimum. In the general case, we cannot make assumptions about whether the embedding or the embedded node is more trustworthy. We resolve the conflict as follows: if the child element requests a larger size than permitted by the parent element, we draw the child element with the smaller size, visually indicate to the user that the child could not be fully displayed, and allow the user to manually maximize it. This solution can be seen as a sort of generalization of the behavior of typical browser popup-blockers.

Here, we have discussed a general design for maintaining our interface layout tree invariants and for handling size conflicts; we refine these techniques based on our implementation experiences in Section 4.6. For example, we find that the tree transformation approach can be eliminated or simplified in practice.

4.4.2 Maintaining Developer Flexibility

In designing a security-aware user interface toolkit, we aim to support these security properties alongside the traditional goals of minimizing the difficulty of implementing typical interfaces (i.e., minimizing a toolkit's threshold [121]) while still providing maximal expressiveness and flexibility (i.e., maximizing a toolkit's ceiling [121]).

Although isolating elements by principal and maintaining the described tree invariants supports our desired security properties, naively implementing these mechanisms introduces undesired restrictions on interface flexibility. To restore this necessary flexibility, we now extend this basic architecture to allow developers to expose model-level APIs, compose elements, and display feedback across principals.

Model-Level Event Listeners

In traditional user interface toolkits, interface elements allow applications to attach listeners for various events (e.g., click, key, touch). For sensitive elements, however, these generic listeners may present security risks. For example, an application can use an `onKeyListener` to eavesdrop as a user enters a password into a third-party federated login element. To prevent such attacks, elements in our architecture default to restricting such listener hooks to callers of the same principal (or the system itself).

However, recall that elements may wish to expose certain events across principals (e.g., to inform an application when an ad has loaded, to provide a captured photo, to update detected GPS coordinates). Unlike generic event listeners, these events have model-level semantics (i.e., they are meaningful at the application level, not at the interface element level). We thus apply *model-level event listeners*, which can be used by a sensitive element to expose higher-level data or events to the principal that embeds it. For example, a payment library might now allow applications to attach arbitrary listeners to a payment dialog, but could explicitly expose events for successful payment. Existing UI toolkits use model-level events to provide meaningful notifications related to manipulation of an element (e.g., `ItemListeners` on menu items and `ChangeListeners` on sliders), but security-sensitive interface elements are likely to expose even higher-level events than current examples.

In our architecture, both generic and model-level event listeners execute in the principal of the defining code, not that of the element to which they are attached. Otherwise, an attacker could inject arbitrary code into another principal's code (e.g., by attaching an `onClickListener` to an element belonging to the principal of the attacker's choice).

Composition Across Principals

We have thus far considered only the *visual* composition of elements belonging to different principals. However, interface designers require greater flexibility to *logically* compose elements. For example, recall from Chapter 3 the discussion of composed access control gadgets: a developer may wish to design a button that takes location-tagged photos, combining a location ACG with a camera ACG. Conceptually, the developer would like to embed icons

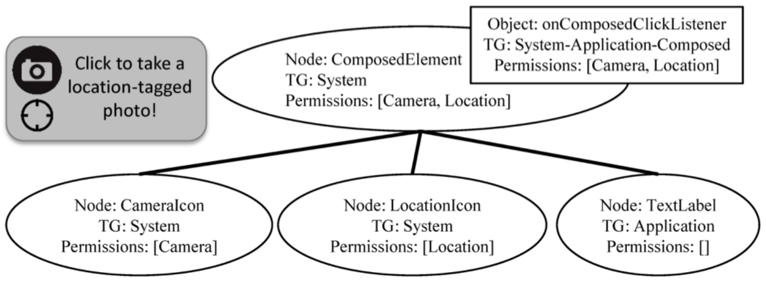


Figure 4.3: **Composition Across Principals.** In this example, the application has composed the system-trusted camera and location icons with its own label to create a custom button that has both camera and location permissions.

from both ACGs into a custom element and inherit the former's principals and permissions.

We support composition of elements from multiple principals by introducing a system-defined ComposedElement. Any principal may choose to expose composable sub-elements for inclusion in a ComposedElement (e.g., the system may expose a User Account Control shield icon with the "administrator" permission or a GPS icon with the "location" permission). Another developer can then mash up these sub-elements with custom elements in a ComposedElement. For instance, Figure 4.3 shows a composed button for taking a location-tagged photo.

Our toolkit must ensure that allowing compositions does not violate the security properties achieved in previous sections. In particular, a ComposedElement must not allow code to manipulate or observe elements belonging to another principal or to inappropriately access restricted APIs. In order to retain the benefits of the layout tree invariants previously described, we assign ComposedElements to the "system" principal. ComposedElements are thereby allowed to contain sub-elements of other principals. Our invariants continue to hold within the ComposedElement. This ensures, for example, that an application cannot use a composition to eavesdrop on a third-party login field.

Recall the goal of a composition is for the resulting element to inherit the permissions of its constituent elements. However, we do not wish to allow applications to inject arbitrary code into a ComposedElement, as this code will run in the system's principal.

To achieve these goals simultaneously, the ComposedElement allows the principal that embeds it to attach event listeners. Unlike other event listeners (which run in the attacher's principal), listeners of a ComposedElement run in a temporary principal derived from the ComposedElement's sub-elements; its set of permissions is the union of their permissions. For example, the ComposedElement in Figure 4.3 includes location and camera sub-elements, resulting in an `onComposedClickListener` with principal and permissions defined as {System-Application-Composed, [Location, Camera]}. The attached listener can thus access the necessary system APIs (but not other system APIs).

Without an additional timeout mechanism, such ComposedElement listeners cannot be prevented from running and taking advantage of these permissions indefinitely. Thus, if a principal wishes to prevent this risk for certain permissions, it should not expose them via composable sub-elements. For example, if the system wishes to grant camera access only via photos returned directly from a system-trusted camera button (i.e., a camera access control gadget), it should not (and does not need to) expose a composable camera icon.

Flexibility of Feedback

Isolating interface elements from different principals as described thus far will restrict developer flexibility in displaying feedback that requires access to elements and data across principals. We examine drag-and-drop and lenses as canonical examples of such flexible feedback, and we describe how our toolkit architecture preserves developer flexibility for these types of scenarios.

During a drag-and-drop operation, dragging an object over a potential drop target often yields feedback indicating whether it can accept the drop and possibly what effect the drop will have. This feedback may require access to the contents of the drag object (not just its type). For example, a text editor may wish to show what dropped text will look like in the current font before the user completes the drop.

However, the drop target may belong to a different principal than the drag object. Until a user drops the object, it is not clear that the potential drop target is the intended recipient, so it should not receive full access to the drag object. Providing such access would allow a malicious non-target application to steal potentially sensitive information. A challenge for our toolkit architecture is therefore to allow the potential drop target to display feedback that relies upon the content of the drag object.

Similarly, lenses [19] are overlaid on an interface to display flexible feedback about the underlying elements. For example, a lens over a set of map tiles might magnify the underlying map features or highlight certain cities. However, the elements from which a lens requires information in order to paint itself may not all belong to the lens's principal. System-trusted lenses can have full access, but supporting arbitrary lenses requires allowing the lens element to show feedback based on elements belonging to other principals. As with drag and drop, we wish to support feedback in the lens without granting the lens's principal full access to the underlying interface elements.

Supporting flexible feedback. When an element wishes to display this type of cross-principal feedback, the system launches a new sandbox that can run and isolate arbitrary code, preventing it from communicating over the network or with other applications (note that our toolkit design is independent of the implementation of this sandbox). Isolated in this way, the system executes feedback generation code provided by the element in question. This code generates feedback by accessing and manipulating a copy of the layout tree and any other restricted data needed to generate appropriate feedback (e.g., the drag object). However, the feedback code cannot break Data Isolation because it cannot communicate outside of the sandbox. It also cannot violate Display Integrity, as it manipulates only a copy of the restricted data.

The feedback code produces a temporary version of the relevant portion of the layout tree. The system displays this feedback in a system-trusted overlay element, thus never granting the original element access to the sensitive data. Note that it is possible for malicious feedback code to show inaccurate feedback (e.g., a lens that displays cities not on the underlying map). Thus, while users may interact with the feedback element (e.g., clicking on the map inside the lens), this feedback is not automatically propagated to the underlying elements. If desired, these elements can expose methods allowing for feedback event propagation.

Android prototype implementation. Our original paper on a security-aware UI toolkit architecture [136] contained a prototype Android implementation that differs from Layer-Cake, which we describe in this chapter (Section 4.6). We mention the original prototype here, because (unlike LayerCake) it included an implementation of flexible feedback for drag-and-drop, as described above. We refer to our 2012 paper for the details of that implementation [136].

4.5 The Case for Secure UIs in Android

While Section 4.4 developed techniques for a user interface toolkit architecture in general, we now specifically make the case for secure embedded UIs in Android. The fact that an Android application cannot embed another application's interface results in a fundamental trust assumption built into the Android UI toolkit. In particular, every UI element trusts its parent and its children, who each have unrestricted access to the element's APIs. Vulnerabilities arise when this trust assumption is violated, e.g., because an embedded element is provided by a third-party library.

We now introduce several case studies illustrating that embedded user interface scenarios in stock Android are often either insecure or impossible. We will return to these case studies in Section 4.7 and reevaluate them in the context of our implementation.

Advertising. In stock Android, applications wishing to embed third-party advertisements must include an ad library, such as AdMob or Mobclix, which runs in the embedding application's process. These libraries provide a custom UI element (an AdView) that the embedding application instantiates and embeds. As has been discussed extensively in prior work [131, 148], the library model for third-party advertisements comes with a number of security and privacy concerns. For example, the host application must trust the advertising library not to abuse the host's permissions or otherwise exploit a buggy host application. Additionally, ad libraries ask their host applications to request permissions (such as location and Internet access) on their behalf; applications that request permissions not clearly relevant to their stated purpose can desensitize users to permission warnings [50].

We have also identified and experimentally demonstrated threats to the AdView [136].

Parent applications can mount a programmatic clickfraud attack in which they programmatically click on embedded ads to increase their advertising revenue. Similarly, parent applications can mount clickjacking attacks by, for example, covering the AdView with another UI element that does not accept (and thus lets pass through) input events.

WebViews. One of the built-in UI elements provided by Android is the WebView, which can load local HTML content or an arbitrary URL from the Internet. Though WebViews appear conceptually similar to iframes, they do not provide many of the same security properties. In particular, WebViews — and more importantly, the contained webpage — can be completely manipulated by the containing application, which can mount attacks including programmatic clicking, clickjacking, and input eavesdropping [98]. Thus, for example, if an Android application embeds a WebView that loads a login page, that application can eavesdrop on the user's password as he or she enters it into the WebView.

Facebook social plugins. On the Web, Facebook provides a set of social plugins [46] to third-party web developers. These plugins include the "Like" button, a comments widget, and a feed of friends' activities on the embedding page (e.g., which articles they liked or shared). These social plugins are generally implemented as iframes and thus isolated from the embedding page.

While Facebook also supplies an SDK for smartphones (iOS and Android), this library — like all libraries, it runs in the host application's process — does not provide embeddable plugins like those found on the Web. A possible reason for this omission is that Facebook's SDK for Android cannot prevent, for example, applications from programmatically clicking on an embedded "Like" button or extracting private information from a recommendations plugin. Although developers can manually implement a social plugin using a WebView, this implementation suffers from the security concerns described above. Thus, though embeddable social plugins on mobile may be desirable to Facebook, they cannot be achieved securely on stock Android.

Access control gadgets. Finally, Chapter 3 proposed access control gadgets (ACGs), secure embedded UI elements that are used to capture a user's permission-granting intent

Category	*Security Requirement*	*(Section Number) Approach*
Display Integr.	Prevents direct modification	*(4.6.3)* Embedded elements in isolated, overlaid windows.
	Prevents size manipulation	*(4.6.6)* User notifications on size conflicts.
Input Integr.	Prevents programmatic input	*(4.6.3)* Embedded elements in isolated, overlaid windows.
Intent Integr.	Clickjacking protection	*(4.6.7)* No input delivered if view/window not fully visible.
	Prevents input DoS	*(4.6.3)* Embedded windows attached to system root.
Data Isolation	Prevents access to display	*(4.6.3)* Embedded elements in isolated, overlaid windows.
	Prevents input eavesdropping	*(4.6.3)* Embedded windows attached to system root.
	Prevents focus stealing	*(4.6.4)* Focus changes only in response to real user clicks.
	Prevents ancestor redirection	*(4.6.8)* Prompts and *(4.7.2)* redirection ACG.
UI-API Links	APIs can verify caller	*(4.6.2)* Elements from different principals run in separate calling processes (identifiable by package name).

Table 4.2: **Techniques for Secure Embedded UI.** This table summarizes how LayerCake (our modified version of Android 4.2) achieves each of the desired security properties for embedded user interface elements.

(e.g., to grant an application access to the user's current location). Authentically capturing a user's intent relies on a set of UI-level security properties including clickjacking protection, display isolation, and user intent protection. As we describe in this chapter, fundamental modifications to Android are required to enable secure embedded elements like ACGs.

4.6 LayerCake: Secure Embedded UIs for Android

We now explore what it takes to support secure embedded UIs, under the definitions from Section 5.2, in the Android framework. As no existing Android-based solutions meet these goals, we view this implementation as an opportunity to consider secure embedding from scratch. While we adapt techniques from prior work, we find that previously published guidelines are not always directly applicable. For example, we found that we could simplify an approach from Section 4.4.1 when overlaying cross-application content, but that we faced additional practical challenges, such as the need to propagate layout changes and handle multiple levels of nesting. We further discuss these and other challenges and lessons in Section 4.9.

We thus created *LayerCake*, a modified version of the Android framework that supports cross-application embedding via changes to the ActivityManager, the WindowManager, and input dispatching. We added or modified 2400 lines of source code across 50 files in Android 4.2 (Jelly Bean). Table 4.2 summarizes the implementation choices that achieve our desired security properties.

We have made the LayerCake source code, system images, SDK, and sample applications available at `https://layercake.cs.washington.edu/`.

4.6.1 Android Background

Android user interfaces are focused around Activities, which present the user with a particular view (or screen) of an application. An application generally consists of multiple Activities (e.g., settings, comments, and newsfeed Activities), each of which defines an interface consisting of built-in or custom UI elements (called Views).

Android's ActivityManager keeps only one Activity in the foreground at a time. An application cannot embed an Activity from another application, and two applications cannot run side-by-side. While Android does provide support for ActivityGroups (deprecated in favor of Fragments) to improve UI code reuse within an application, these mechanisms do not provide true Activity embedding and are not applicable across application and process boundaries. The goal of our exploration is to allow one application to embed an Activity from another application (running in that other application's process).

Each running Android application is associated with one or more windows, each of which serves as the root of an interface layout tree consisting of application-specified Views. Android's WindowManager isolates these windows from each other — e.g., an application cannot access the status bar's window (shown at the top of the screen) — and appropriately dispatches user input. Our implementation relies on these isolation properties.

While only one Activity can be in the foreground, multiple applications/processes may have visible windows. For example, the status bar runs in the system process, and the window of one application may be visible below the (partially) transparent window of another. As an example of the latter, AdSplit [148] achieves visual embedding by taking advantage of an application's ability to make portions of its UI transparent. However, recall from

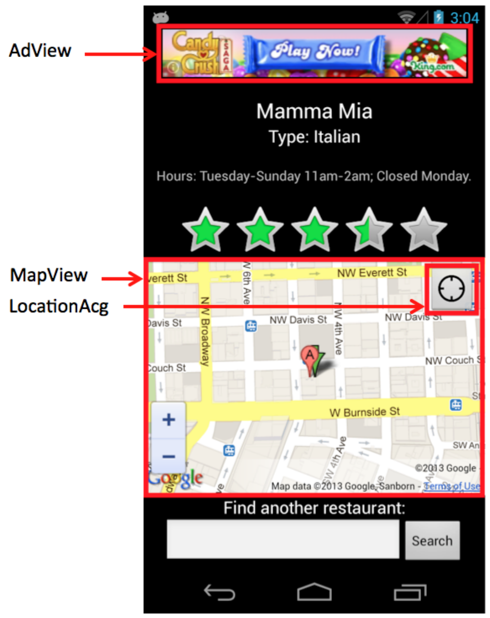


Figure 4.4: **Sample Application.** This restaurant review application embeds two third-party Activities, an advertisement and a map. The map Activity further embeds an access control gadget (ACG) for location access.

Section 4.3 and Table 4.1 that this approach is insufficient for generalized embedded UI security.

4.6.2 Supporting Embedded Activities

LayerCake introduces a new View into Android's UI toolkit (Java package `android.view`) called `EmbeddedActivityView`. It allows an application developer to embed another application's Activity within her application's interface by specifying in the parameters of the EmbeddedActivityView the package and class names of the desired embedded Activity. Figure 4.4 shows a sample application that embeds several Activities.

We extended Android's ActivityManager (Java) to support embedded Activities, which are launched when an EmbeddedActivityView is created and displayed. Unlike ordinary Activities, embedded Activities are not part of the ActivityManager's task stack or history list, but rather share the fate of their parent Activity. Crucially, this means that an embedded Activity's lifecycle is linked to that of its parent: when the parent is paused, resumed, or destroyed, so are all of its embedded children.

An Activity may embed multiple other Activities, which themselves may embed one or more Activities (multiple nesting). Each embedded Activity is started as a new instance,

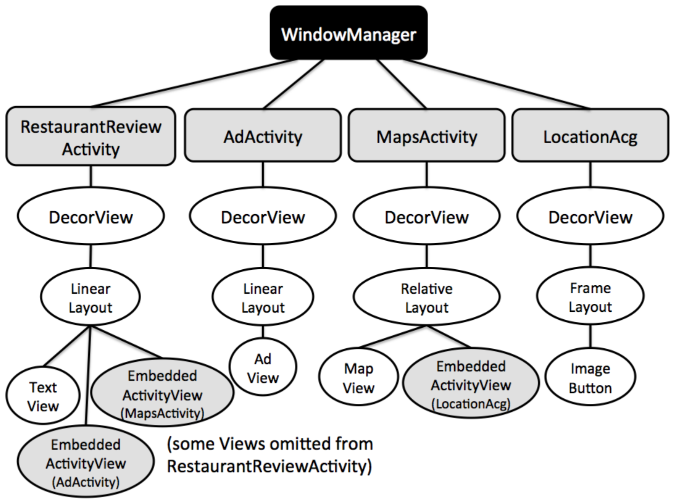

WindowManager
RestaurantReview Activity
AdActivity
MapsActivity
LocationAcg
DecorView
DecorView
DecorView
DecorView
Linear Layout
Linear Layout
Relative Layout
Frame Layout
Text View
Embedded ActivityView (MapsActivity)
Ad View
Map View
Embedded ActivityView (LocationAcg)
Image Button
Embedded ActivityView (AdActivity)
(some Views omitted from RestaurantReviewActivity)

Figure 4.5: **Window Management.** This figure shows the Window/View tree for the Activities in Figure 4.4. Embedded Activities are not embedded in the View tree (circles) of their parent, but rather within a separate window. The WindowManager keeps track of a window (grey squares) for each Activity and visually overlays an embedded window on top of the corresponding EmbeddedActivityView in the parent Activity.

so multiple copies of the same Activity are independent (although they run in the same application, allowing changes to the application's global state to persist across different Activity instances).

4.6.3 Managing Windows

Properly displaying embedded Activities required modifications to the Android Window-Manager (Java). One option for achieving embedded UI layouts is to literally nest them — that is, to add the embedded Activity's Views (UI elements) as children in the parent Activity's UI tree. However, this design would allow the parent Activity to mount input eavesdropping and denial-of-service attacks on the child Activity. Thus, following the interface layout tree invariants described in Section 4.4.1 we do not literally nest the interface elements of embedded Activities inside the parent Activity. Instead, an embedded Activity is displayed in a new window, overlaid on top of the window to which it is attached (i.e.,

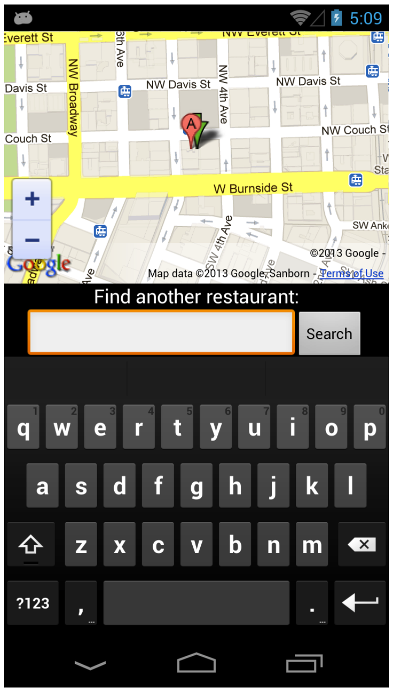


Figure 4.6: **Panning for Software Keyboard.** The restaurant review application (from Figure 5.5), including its overlaid embedded windows, must be panned upward to make room for the software keyboard underneath the in-focus text box.

the window of the parent Activity). This overlay achieves the same visual effect as literal embedding but prevents input manipulation attacks. Figure 4.5 shows an example of the interface layout trees associated with the Activities in the sample application in Figure 4.4. We note that we were able to simplify the approach from Section 4.4.1, which we found in our implementation experience to be overly general (see Section 4.9).

By placing embedded Activities into their own windows instead of into the parent's window, we also inherit the security properties provided by the isolation already enforced by the WindowManager. In particular, this isolation prevents a parent Activity from modifying or accessing the display of its child Activity (or vice versa).

The relative position and size of an overlaid window are specified by the embedding application in the layout parameters of the EmbeddedActivityView and are honored by the WindowManager. (Note that the specified size may violate size bounds requested by the embedded Activity, as we discuss in Section 4.6.6.)

The layout parameters of an embedded Activity's window must remain consistent with those of the associated EmbeddedActivityView, a practical challenge not described in prior work. For example, when the user reorients the phone into landscape mode, the parent Activity will adjust its UI. Similarly, when the soft keyboard is shown, Android may pan the Activity's UI upwards in order to avoid covering the in-focus text box with the keyboard (Figure 4.6). In both cases, the embedded Activity's windows must be relocated appropriately. To support these dynamic layout changes, the EmbeddedActivityView reports its layout changes to the WindowManager, which applies them to the associated window.

Finally, since LayerCake supports multiple levels of embedding, it must appropriately display windows multiple levels down (e.g., grandchildren of the top-level Activity). For example, suppose ActivityA embeds ActivityB which embeds ActivityC. If the EmbeddedActivityView (inside ActivityB) that corresponds to ActivityC is not fully visible — e.g., because it is scrolled halfway out of ActivityB's visible area — then the window corresponding to ActivityC must be cropped accordingly (Figure 4.7). This cropping is necessary because ActivityC is not literally nested within ActivityB, but rather overlaid on top of it, as discussed above.

4.6.4 Handling Focus

Both the parent and any embedded Activities must properly receive user input. While touch events are dispatched correctly even in the presence of visually overlapping windows, stock Android grants focus for key events only to the top-level window. As a result, only the window with the highest Z-order in an application with embedded Activities will ever receive key events. We thus modified Android to switch focus between windows belonging to the parent or any embedded Activities within an application, regardless of Z-order.

In particular, we changed the input dispatcher (C++) to deliver touch events to the WindowManager in advance of delivering them to the resolved target. When the user touches an unfocused window belonging to or embedded by the active application, the WindowManager redirects focus. Windows that might receive the redirected focus include that of the parent Activity, the window of any embedded Activity, or an attached window from the same process (e.g., the settings context menu, which Android displays in a new window). Switching focus only in response to user input (rather than an application's request)

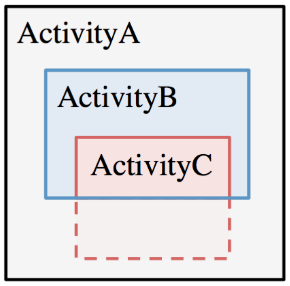


Figure 4.7: **Cropping Further Nested Activities.** If a grandchild (ActivityC) of the top-level Activity (ActivityA) is placed or scrolled partly out of the visible area of its immediate parent (ActivityB), it must be cropped accordingly.

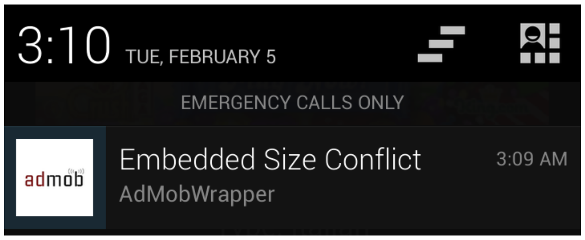
prevents a parent or child window from stealing focus to eavesdrop on input intended for another principal.

4.6.5 Supporting Cross-Principal APIs

To support desired functionality, embedded UI elements and their parents must communicate. For example, an application embedding an ad may wish to communicate keywords to the ad provider, or a system-defined location button (ACG) may wish to pass the current location to the parent application in response to a user click. To enable flexible communication between embedded Activities and their parents, we leverage the Android Interface Definition Language (AIDL), which lets Android applications define interfaces for inter-process communication. We thus define the following programming model, enabling the model-level event listeners described in Section 4.4.2.

Each embeddable Activity defines two AIDL interfaces, one that it (the child) will implement, and one that the parent application must implement. For example, the advertisement (child) may implement a `setKeywords()` method, and the ad's parent application may be asked to implement an `onAdLoaded()` method to be notified that an ad has been successfully loaded. When an application wishes to embed a third-party Activity, it must keep copies of the relevant interface files in its own source files (as is standard with AIDL), and it must implement `registerChildBinder()`. This function allows the child Activity, once started, to make a cross-process call registering itself with the parent.

We note that this connection is set up automatically only between parents and immediate children, as doing so for siblings or farther removed ancestors may leak information about



Embedded Size Conflict
AdMobWrapper
3:09 AM

Figure 4.8: **Size Conflict Notification.** If the AdMobWrapper application specifies a minimum size that the RestaurantReviewActivity does not honor when it embeds the advertisement, a system notification is displayed to the user. Clicking on the notification displays a full-screen advertisement Activity.

the UIs embedded by another principal.

4.6.6 Handling Size Conflicts

Recall from Section 4.6.3 that the WindowManager honors the parent application's size specification for an EmbeddedActivityView. This policy prevents a child element from taking over the display (a threat discussed further in the context of ancestor redirection below). However, we also wish to prevent size manipulation by the parent.

We observe that it is only of concern if an embedded Activity is given a smaller size than requested, since it need not scale its contents to fill its (possibly too large) containing window. Thus, we modified the Activity descriptors to include only an optional minimum height and width (specified in density-independent pixels).

Section 4.4.1 described different size conflict policies based on whether the embedded element is trusted or untrusted by the system. If it is trusted (e.g., a system-defined ACG), its own size request should be honored; if it is untrusted (e.g., an ad that requests a size filling the entire screen), the parent's size specification is honored. However, we observe in our implementation experience that a malicious parent can mimic the effect of making a child element too small using other techniques, such as scrolling it almost entirely off-screen — and that doing so maliciously is indistinguishable from legitimate possible scroll placements. We thus further consider the failure to meet minimum size requirements in the context of clickjacking (Section 4.6.7).

Thus, since enforcing a minimum size for trusted embedded elements does provide additional security properties in practice, we use the same policy no matter whether mis-sized elements are trusted or untrusted by the system. That is, the WindowManager honors the size specifications of the parent Activity. If these values are smaller than the embedded Activity's request, a status bar notification is shown to the user (Figure 4.8). Similar to a browser's popup blocker, the user can click this notification to open a full-screen (non-embedded) version of the Activity whose minimum size was not met.

4.6.7 Support for Clickjacking Prevention

In a clickjacking attack [70], a malicious application forces or tricks a user into interacting with an interface, generally by hiding important contextual information from the user. For instance, a malicious application might make a sensitive UI element transparent or very small, obscure it with another element that allows input to pass through it, or scroll important context off-screen (e.g., the preview associated with a camera button).

To prevent such attacks, an interface may wish to discard user input if the target is not fully visible. Since it may leak information about the embedding application to let an element query its own visibility, LayerCake allows embedded Activities to request that the Android framework simply not deliver user input events if the Activity is:

1. *Covered (fully or partly) by another window.* This request is already supported by stock Android via `setFilterTouchesWhenObscured()`.
2. *Not the minimum requested size.* A parent application may not honor a child's size request (see Section 4.6.6).
3. *Not fully visible due to window placement.* An embedded Activity's current effective window may be cropped due to scrolling.

Note that an embedded Activity need not be concerned about a malicious parent making it transparent, because stock Android already does not deliver input to invisible windows. Similarly, an Activity need not be concerned about malicious visibility changes to UI elements within its own window, since process separation ensures that the parent cannot manipulate these elements. To prevent timing-based attacks, these criteria should be met for some minimum duration [70] before input is delivered, a check that we leave to future work.

We emphasize that embedded iframes on the Web today can neither discover if all of these criteria are met — due to the same-origin policy, they cannot know if the parent page has styled them to be invisible or covered them with other content — nor request that the browser discard input under these conditions.

4.6.8 Preventing Ancestor Redirection

Android applications use Intents to launch Activities either in their own execution context (e.g., to switch to a Settings Activity) or in another application (e.g., to launch a browser pointed at a specified URL). In response to a `startActivity(intent)` system call, Android launches a new top-level full-screen Activity. Recall that allowing an embedded element to redirect the ancestor UI without user consent is a security concern.

We thus make two changes to the Android framework. First, we introduce an additional flag for Intents that starts the resulting Activity inside the window of the embedded Activity that started it. Thus, for example, if an embedded music player wishes to switch from its MusicSelection Activity to its NowPlaying Activity without breaking out of its embedded window, it can do so by specifying `Intent.FLAG_ACTIVITY_EMBEDDED`. (If the music player is not embedded, this flag is simply ignored.)

Second, we introduce a prompt shown to users when an embedded Activity attempts to launch another Activity full-screen (i.e., not using the flag described above). This may happen either because it is a legacy application unaware of the flag, or for legitimate reasons (e.g., a user's click on an embedded advertisement opens a new browser window). However, studies have shown that prompting users is disruptive and ineffective [118]; in Section 4.7.2 we discuss an access control gadget that allows embedded applications to launch full-screen Intents in response to user clicks without requiring that the system prompt the user.

4.7 Case Studies

We now return to the case studies introduced in Section 4.5 and describe how LayerCake supports these and other scenarios. Table 4.3 shows that implementation complexity is low, especially for parent applications.

4.7.1 Geolocation ACG

To support user-driven access for geolocation, we implemented a geolocation access control gadget (ACG) in the spirit of Chapter 3.We added a `LocationAcg` Activity to Android's SystemUI (which runs in the system process and provides the status bar, the recent applications list, and more). This Activity, which other applications can embed, simply displays a location button (see Figure 4.4).

Following a user click, the SystemUI application, not the parent application, accesses Android's location APIs. To then receive the current location, the parent application must implement the `locationAvailable()` method defined in the parent AIDL interface provided by the LocationAcg's developers (us).

Security discussion. LayerCake provides the security properties required to enable ACGs. In particular, the parent application of a LocationAcg cannot trick the user into clicking on the gadget, manipulate the gadget's look, or programmatically click on it.

We emphasize again that this ACG provides location information to the parent application only when the user wishes to share that information; a well-behaving parent application will not need location permissions. In a system like Android, where applications can request location permissions in their manifest, it is an open question how to incentivize developers to use the corresponding ACG instead of requesting that permission. Chapter 3 suggested incentives including increased scrutiny at app store review time of applications requesting sensitive permissions.

4.7.2 Redirection Intent ACG

In Section 4.6.8, we introduced a system prompt when an embedded Activity attempts to start a full-screen Activity. However, prompts are known to be disruptive and often

	Lines of Java	**Parent Lines of Java**
Geolocation ACG	111	14
Redirection Intent ACG	75	23
Secure WebView	133	13
Advertisement	562	37
FacebookWrapper	576	30

Table 4.3: **Implementation Complexity.** Lines of code for (1) the embedded Activity and (2) the parent's implementation of the AIDL interface. We omit legacy applications because they required no modifications and expose no parent interfaces. Implementation complexity is low, especially for embedders.

ignored, especially following a user action intended to cause the effect about which the prompt warns [178]. For example, a user who clicks on an embedded ad in stock Android today expects it to open the ad's target in a new (non-embedded) browser window. Following the philosophy of user-driven access control (Chapter 3) we thus allow embedded Activities to start top-level Activities without a prompt if `startActivity()` is called in response to a user's click.

To verify that the user has actually issued the click, we take advantage of our system's support for ACGs and implement an ACG for top-level redirection. This `RedirectAcg` Activity again belongs to Android's SystemUI application. It consists primarily of an ImageView that may be filled with an arbitrary Bitmap, allowing the embedder to completely specify its look. An embedded Activity that embeds such an ACG (two levels of embedding) thus uses the cross-process API provided by the RedirectAcg to (1) provide a Bitmap specifying the look, and (2) specify an Intent to be supplied to the `startActivity()` system call when the user clicks on the RedirectAcg (i.e., the ImageView's `onClick()` method is fired).

Security discussion. The UI-level security properties provided by LayerCake ensure that the RedirectAcg's `onClick()` method is fired only in response to real user clicks. In other words, the embedding application cannot circumvent the user intent requirement for launching a top-level Activity by programmatically clicking on the RedirectAcg or by tricking the user into clicking on it.

Unlike the LocationAcg, however, the embedding application is permitted to fully control the look of the RedirectAcg. This design retains backwards compatibility with the stock Android experience and relies on the assumption that a user's click on anything within an embedded Activity indicates the user's intent to interact with that application. However, alternate designs might choose to restrict the degree to which the redirecting application can customize the RedirectAcg's interface. For example, the system could place a visual "full-screen" or "redirect" indicator on top of the application-provided Bitmap, or it could simply support a stand-alone "full-screen" ACG that applications wishing to open a new top-level view must display without customization.

Note that developers are incentivized to use the RedirectAcg because otherwise attempts to launch top-level Activities will result in a disruptive prompt (Section 4.6.8).

4.7.3 Secure WebView

We implemented a SecureWebView that addresses security concerns surrounding Android WebViews [97, 98]. The SecureWebView is an Activity in a new built-in application (WebViewApp) that consists solely of an ordinary WebView (inside a FrameLayout) that fills the Activity's whole UI. Thus, when another Activity embeds a SecureWebView, the internal WebView takes on the dimensions of the associated EmbeddedActivityView.

The SecureWebView Activity exposes a safe subset (see below) of the underlying WebView's APIs to its embedding process. The current version of LayerCake exposes only a subset of these APIs for demonstration purposes. A complete implementation will need to properly (de)serialize all complex data structures (e.g., `SslCertificate`) across process boundaries.

Security discussion. Separating out the Android WebView into another process — that of the WebViewApp — provides important missing security properties. It is no longer possible to eavesdrop on input to the embedded webpage, to extract content or programmatically issue input, or to manipulate the size, location, or transparency of the WebView to mount clickjacking attacks.

While the SecureWebView wraps the existing WebView APIs, it should avoid exposing certain sensitive APIs, such as those that mimic user input (e.g., scrolling via `pageUp()`) or that directly extract content from the WebView (e.g., screenshot via `capturePicture()`). Note, however, that APIs which redirect the SecureWebView to another URL are permitted, as the parent application could simply open a new SecureWebView instead.

Ideally, Android would replace the WebView with the SecureWebView, but this change would not be backwards compatible and may conflict with the goals of some developers in using WebViews. Thus, we observe that using a SecureWebView also benefits the embedding application: if it exposes an API to the webpage via an ordinary WebView (using `addJavascriptInterface()`), a malicious page could use this to manipulate the host application. Process separation protects the host application from such an attack, and since the WebViewApp has only the `INTERNET` permission, the attack's effect is limited. Additionally, WebView cookies are not shared across processes; the SecureWebView allows applications to reuse (but not access) existing cookies, possibly providing a smoother user experience.

4.7.4 Advertisements

Recall that stock Android applications embedding third-party advertisements include an ad library that runs in the host application's process and provides an AdView element. Our modifications separate the AdView out into its own process (see the advertisement in Figure 4.4). To do this, we create a wrapper application for the AdMob advertising library [57]. The wrapper application exposes an embeddable Activity (called `EmbeddedAd`) that instantiates an AdMob AdView with the specified parameters. This Activity exposes all of AdMob's own APIs across the process boundary, allowing the embedding application to specify parameters for the ad.

Security discussion. Moving ads into their own process (one process per ad library) addresses a number of the concerns raised in Section 4.5. In particular, an ad library can no longer abuse a parent application's permissions or exploit a buggy parent application. Furthermore, the permissions needed by an ad library, such as Internet and location permissions, must no longer be requested by the parent application (unless it needs these permissions for other purposes).

Note that all ads from a given ad library — even if embedded by different applications — run in the same process, allowing that ad application to leverage input from different embedders. For example, if one application provides the user's age and another provides the user's gender, the ad application can better target ads in *all* parent applications, without revealing additional information to applications that did not already have it. (However, we note that some users may prefer that ad applications not aggregate this information.)

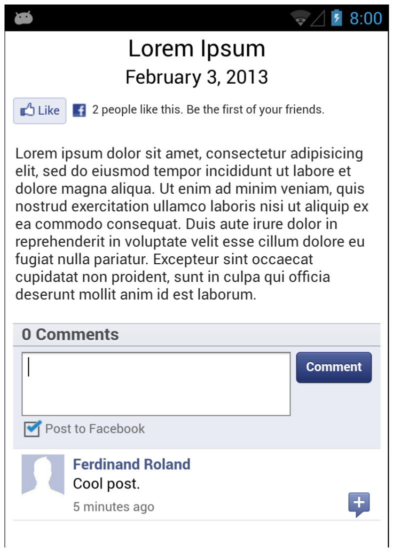
LayerCake goes beyond process separation, providing UI-level security absent in most prior systems (except AdSplit [148]). Most importantly, the parent can no longer mount programmatic click fraud attacks.

4.7.5 Facebook Social Plugins

We can now support embedded Facebook social widgets in a secure manner. We achieve this by creating a Facebook wrapper application that exposes Activities for various Facebook social widgets (e.g., a Comments Activity and a Like Activity — see Figure 4.9). Each Activity displays a WebView populated with locally-generated HTML that references the Facebook JavaScript SDK to generate the appropriate plugin (as done ordinarily by web pages and as specified by Facebook [46]).

Security discussion. LayerCake supports functionality that is impossible to achieve securely in stock Android and may be desirable to Facebook. This functionality was previously available only on the Web, due to the relative security of embedded iframes (though clickjacking, or "likejacking", remains a problem on the Web). Our implementation protects the social widgets both by separating them into a different process (preventing data extraction, among others), and by enforcing other UI-level security properties (preventing clickjacking and programmatic clicking).

We observe that a malicious application might attempt to mimic the FacebookWrapper application by populating a local WebView with the HTML for a social plugin. To prevent this attack, we recommend that the FacebookWrapper application include a secret token in the HTML it generates (and that Facebook's backend verify it), similar in approach to CSRF protections on the Web.



Lorem Ipsum

February 3, 2013

2 people like this. Be the first of your friends.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

0 Comments

Ferdinand Roland
Cool post.
5 minutes ago

Figure 4.9: **Facebook Social Plugins.** This example blog application embeds both a Facebook "Like" button and a comments feed, both running in our FacebookWrapper application.

4.7.6 Legacy Applications

The applications discussed so far needed wrapper applications because the wrapped functionality was not previously available in a stand-alone fashion. However, this need is not fundamental — any legacy Android application (i.e., one that targets older versions of the Android SDK) can be embedded using the same techniques.

To demonstrate this, we created an application that embeds both the existing Pandora application and the existing Amazon application. To do so, we needed to discover the names of the corresponding Activities in the existing applications. This information is easy to discover from Android's standard log, which prints information about Intent targets when they are launched. Figure 4.10 shows a screenshot of the resulting application.

Security discussion. As in previous case studies, the embedded Activities are isolated from the parent. Thus, they cannot access sensitive information in or manipulate the UI or APIs in the parent application, or vice versa.

Legacy applications naturally do not use the new `FLAG_ACTIVITY_EMBEDDED` flag when launching internal Activities. While updated versions of Pandora and Amazon could use this flag to redirect within an embedded window, the experience with unmodified legacy applications is likely to be disruptive. Thus, a possible policy (perhaps subject to a user

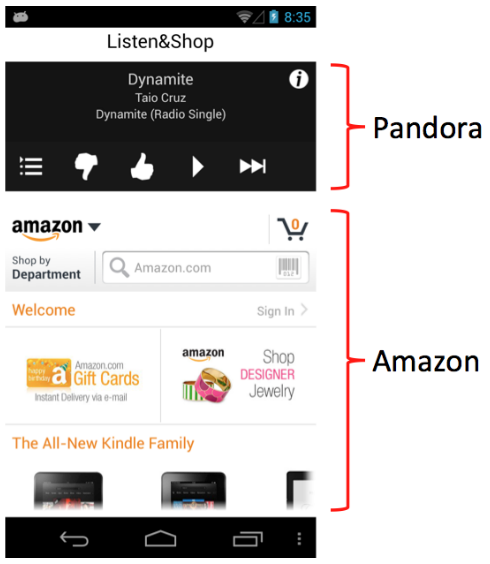


Figure 4.10: **Embedded Pandora and Amazon Apps.** Legacy applications can also be embedded, raising policy questions regarding top-level intents and embedding permissions.

preference setting) for such applications is to internally modify all Activity launches to use the new flag, never allowing these applications to break out of their embedded windows.

Embedding arbitrary applications that were not intended by their developers to be embedded also raises the question of embedding permissions. Some Activities may wish never to be embedded, or to be embedded only by authorized parents. Future modifications to LayerCake should support such permissions.

4.8 Performance Evaluation

We evaluate the performance impact of our changes to Android by measuring the time it takes to start an application, i.e., the delay between a `startActivity()` system call and the `onCreate()` call for the last embedded Activity (or the parent Activity, if none are embedded). As shown in the top of Table 4.4, applications with embedded Activities take longer to fully start. The reason for this is that the parent Activity's layout must be created (in its `onCreate()`) before child Activities can be identified. Thus, an application with multiple nested Activities (e.g., RestaurantReviewer) requires linearly more time than an application with only one level of nesting (e.g., FacebookDemo or Listen&Shop). We note that the parent Activity's own load time is unaffected by the presence of embedded content (e.g., the FacebookDemo Activity starts in 160 ms, even though the embedded

	Load time (10 trial average)	
Application	*No Embedding*	*With Embedding*
RestaurantReviewer	163.1 ms	532.6 ms
FacebookDemo	157.5 ms	304.9 ms
Listen&Shop	159.6 ms	303.3 ms

Scenario	*Event Dispatch Time (10 trial average)*
Stock Android	1.9 ms
No focus change	2.1 ms
Focus change	3.6 ms

Table 4.4: **Performance.** The top table shows the time it takes for the `onCreate()` method of all included Activities to be called. We note that the time to load the parent Activity remains the same whether or not it uses embedding, so the time for the parent to begin displaying native content is unaffected. The bottom table shows that the effect of intercepting input events in the WindowManager for possible focus changes is minor.

Facebook components require 300 ms). Prior work [117] has argued that the time to display first content is more important than full load time.

We also measure input event dispatch time (e.g., the time it takes for Android to deliver a touch event to an application). Specifically, we evaluate the impact of dispatching input events first to the WindowManager, allowing it to redirect focus if appropriate (Section 4.6.4). The bottom of Figure 4.4 shows that involving the WindowManager in dispatch has a negligible performance impact over stock Android; changing focus has a greater impact, but it is not noticeable by the user, and focus change events are likely rare.

We can also report anecdotally that the effect of embedding on the performance of our case study applications was unnoticeable, except that the panning of embedded windows (for the software keyboard) appears to lag slightly. This case could likely be optimized by batching cross-process relayout messages.

Finally, supporting embedded Activities may result in more applications running on a device at once, potentially impacting memory usage and battery life. The practical impact of this issue depends on the embedding behavior of real applications — for example, perhaps most applications will include ads from a small set of ad libraries, limiting the number of applications run in practice.

4.9 Discussion

Whereas existing systems — particularly browsers — have evolved security measures for embedded user interfaces over time, this chapter has taken a principled approach to defining a set of necessary security properties and building a system with full-fledged support for embedding interfaces based on these properties.

4.9.1 Lessons for Embedded Interfaces

From this process, we provide a set of techniques for systems that wish to support secure cross-application UI embedding. Table 4.2 outlines the security properties provided by LayerCake and summarizes the implementation techniques used to achieve each property. We hope this work, in which we develop security properties and techniques for embedded user interfaces, and bring them together into a practical implementation, will inform the designs and implementations of future systems. We highlight the following lessons:

User-driven ancestor redirection. Embedded applications should not be able to redirect an ancestor application/page without user consent. We argue that a reasonable tradeoff between security and usability is to prompt users only if the redirection attempt does not follow a user click (indicating the user's intent to interact with the embedded content). While newer browsers prevent embedded iframes from redirecting the top-level page programmatically, they do not allow user actions (e.g., clicking on a link with target `_top`) or other mechanisms to override this restriction. In our case studies, we saw that this type of click-enabled redirection can be useful and expected (e.g., when a user clicks on an embedded ad, he or she likely expects to see full-screen content about the advertised product or service). In LayerCake, we were able to apply ACGs in a novel way to capture a user's redirection intent (Section 4.7.2).

Size manipulation as a subset of clickjacking. We initially considered size manipulation (by the parent of an embedded interface element) to be a stand-alone threat. A solution that we considered is to treat elements that are trusted or untrusted by the system differently (e.g., an access control gadget is trusted while an advertisement is not), letting the system enforce the minimum requested size for trusted elements. However, this solution provides no additional security, since a malicious parent can use other techniques to obscure the sensitive element (e.g., partially covering it or scrolling it partly off-screen). Thus, we consider size manipulation as a subset of clickjacking. We suggest that sufficient size be considered an additional criterion (in addition to traditional clickjacking prevention criteria like complete visibility [70, 138]) for the enabling of a sensitive UI element.

Simplification of secure UI layout tree. Section 4.4 proposed invariants for the interface layout tree that ensure a trusted path to every node and describes how to transform an invalid layout tree into a valid one. Our implementation experience shows this solution to be overly general. Embedded elements need not be attached to the layout tree in arbitrary locations; rather, they can always attach to the (system-controlled) root node and overlaid appropriately by the WindowManager (or equivalent). That is, the layout trees of separate principals need never be interleaved, but rather visually overlaid on top of each other, requiring no complex tree manipulations. Simplifying this approach is likely to make it easier and less error-prone for system developers to support secure embedded UI.

4.9.2 New Capabilities

We step back and consider the capabilities enabled by our implementation. In particular, the following scenarios were fundamentally impossible to support before our modifications to Android; LayerCake provides additional security properties and capabilities even beyond the Web, as we detail here.

Isolated Embedded UI. Most fundamentally, LayerCake allows Android applications to securely embed UI running in another process. Conceptually, this aligns the Android application model with the Web model, in which embedded cross-principal content is common. Especially as Android expands to larger devices like tablets, users and application developers will benefit from the ability to securely view and show content from multiple sources in one view.

Secure WebViews. It is particularly important that WebViews containing sensitive content run in their own process. While an Android WebView seems at first glance to be similar to an iframe, it does not provide the security properties to which developers are accustomed on the Web (as discussed in this chapter and identified in prior work [97, 98]). LayerCake matches and indeed exceeds the security of iframes — in particular, a SecureWebView can request that the system not deliver user input to it when it is not fully visible or sufficiently large, thereby preventing clickjacking attacks that persist on the Web.

Access Control Gadgets. Chapter 3 introduced ACGs [138] for user-driven access control of sensitive resources like the camera or location, but that work does not provide concrete guidelines for how the necessary UI-level security properties should be implemented. This chapter provides these details, and we hope that they will guide system developers to include ACGs in their systems. We particularly recommend that browser vendors consider ACGs in their discussions of how to allow users to grant websites access to sensitive resources [170].

4.9.3 Additional Issues

Finally, we discuss unaddressed issues that must be considered in future work.

First is the issue of application dependencies, that is, how to handle the case when an application embeds an Activity from another application that is not installed. Possibilities include automatically bundling and installing dependencies (as also proposed by the authors of AdSplit [148]), giving the user the option of installing the missing application, or simply failing silently. This issue led the authors of AdDroid [131] to decide against running ads in their own process, but we argue that the security concerns of not doing so outweigh this issue. The concern that users might uninstall or replace ad applications to avoid seeing ads could be addressed by giving parent applications feedback when a requested embedded Activity cannot be displayed; applications relying on ads could then display an error message if the required ad library is not available. Updates and differences in library versions required by apps could be handled by Android by supporting multiple installed versions or simply by the ad libraries themselves.

Second is the issue of principal identification: a user cannot easily determine the source of an embedded interface (or even whether anything is embedded). This concern mirrors the Web today, where an iframe's presence or source cannot be easily determined, and we consider this to be an important orthogonal problem.

4.10 Related Work

Finally, we consider additional related work on not discussed inline.

User interface toolkit research generally focuses on reducing the effort needed to develop new interfaces. Because the assumptions embedded in a framework can limit interface designers and hinder the development or adoption of new techniques, research often focuses on achieving maximal expressivity and flexibility for interface developers [36, 38, 40, 71, 121]. For example, prior work has proposed tools to aid in the creation of mashup interfaces on the Web [66, 93, 175]. However, these toolkits are not aimed at addressing security concerns and their implicit trust assumptions can pose challenges to security.

Work by Arthur and Olsen [11] examined protecting user data by separating interface elements based on trust. They provide a methodology that splits an interface across a trusted private device and an untrusted public device (e.g., a connected screen or keyboard) by surfacing a user's trust choices to applications. However, they do not consider embedded user interfaces.

In Section 4.3 and Table 4.1, we explored existing implementations of embedded cross-application user interfaces [44, 131, 136, 148]. These systems have differing goals and employ a variety of techniques, but none fully meets the security requirements defined in [136] and expanded here. In particular, none of these approaches can, without modification, support security-sensitive embedded user interfaces like ACGs [138]. The original ACG implementation built on interface-level security properties provided by the Gazelle browser operating system [172].

Others have explored the problem of clickjacking in more depth. One study [143] found that most framebusting techniques are circumventable, making them ineffective for preventing clickjacking. Other work [70] provides a comprehensive study of clickjacking attacks and defenses, presenting a solution (InContext) that relies on the browser to verify the visual context of sensitive UI elements. LayerCake could be extended to support InContext for additional clickjacking protection.

Our implementation relies on security properties provided by the Android WindowManager. Window system security has been explored previously by such projects as Trusted X [43] (an implementation of the X Window System [54] based on the Compartmented Mode Workstation requirements [176]), a secure version of X based on SELinux [81], and the EROS Trusted Window System [147]. We extend this work by leveraging a secure window system to support secure cross-application UI embedding within a single window.

Other researchers have focused on designing and evaluating interfaces for security-critical interactions, including phishing protection [41], user account control [118], web SSL certificate warnings [158], and social network privacy policies [95]. Our work focuses on securing interface-level threats, and we believe that advances here can be leveraged to help enable research in how to best design security-related interfaces.

4.11 Summary

This chapter has systematically considered the security concerns and requirements for embedded third-party user interfaces, developing a general secure-aware user interface toolkit architecture and then implementing and refining those techniques in a modified version of Android. In doing so, we analyzed existing systems — including browsers, smartphones, and research systems — with respect to these requirements. While browsers have evolved to address many (though not all) of these requirements over time, Android-based implementations have not supported secure embedded interfaces. We thus created LayerCake, a modified version of the Android framework that supports cross-principal embedded interfaces in a way that meets our security goals. The resulting capabilities enable several important scenarios, including advertisement libraries, Facebook social plugins, and access control gadgets, the latter introduced in Chapter 3. Based on our exploration and implementation experience, we provide a concrete set of criteria and techniques that has to date been missing for system developers wishing to support secure interface embedding. This work thereby lays a foundation for tackling the challenge of embedded third-party principals within applications on modern client platforms.

Chapter 5

WORLD-DRIVEN ACCESS CONTROL: PERMISSION GRANTING FOR CONTINUOUS SENSING PLATFORMS

In this chapter, we revisit the second key challenge identified in Chapter 1, application permission granting, in a new context. In Chapter 3, we described user-driven access control as an effective permission granting model for modern operating systems like smartphones. User-driven access control's power comes from the fact that users implicitly indicate the intent to grant permissions in the way that they naturally interact with applications — recall from Chapter 3 that 95% of resource accesses in the studied set of Android applications are coupled with a user's UI interactions. Unfortunately, user-driven access control does not translate well to emerging continuous sensing platforms such as the Microsoft Kinect, Google Glass, or Meta SpaceGlasses, where applications rely on continuous access to sensitive sensor input without frequent user interactions. This chapter introduces *world-driven access control* as a novel permission granting approach for these continuous sensing settings, and lays the groundwork for future research in this space. This work was first described in a 2014 tech report [140].

5.1 Motivation and Overview

Continuous sensing is an emerging technology that enables new classes of applications. New platforms, such as Microsoft Kinect [150], Google Glass [59], and Meta SpaceGlasses [109], fundamentally rely on continuous video and depth cameras to support natural user input via gestures and continuous audio sensing for voice commands. Applications on these platforms leverage these capabilities to deliver new functionality to users. For example, WordLens [134] is a Google Glass and iPhone application that uses the camera to continuously scan for words in the real world. It then shows translations of these words overlaid on the user's vision.

These new capabilities raise serious privacy concerns. Consider a user who enters a

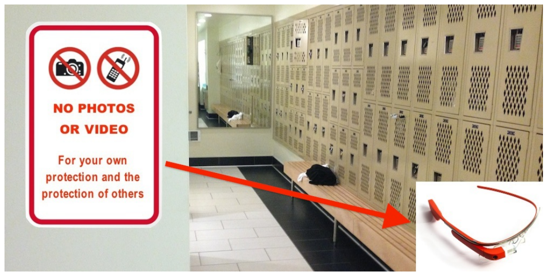


Figure 5.1: **Sensor Privacy Concerns.** Camera restrictions are common in sensitive locations like locker rooms, where both device users' and bystanders' privacy are at risk from untrusted applications. Continuous sensing platforms like Google Glass will make it harder for honest users to mitigate such risks by managing applications' permissions. World-driven access control allows real-world objects to push policies to devices.

locker room while wearing a Google Glass. We identify four classes of privacy concerns in this scenario. First, Word Lens or other *untrusted applications* running on the Glass may see sensitive video data, both about the user and about bystanders. Second, the user may *accidentally record bystanders* by forgetting to turn off the camera while entering the locker room. Third, the user may record herself in a locker room mirror and *accidentally share* the recording on social media. Finally, *malicious users* could use the Glass to record others without their knowledge.

The first three classes of privacy concerns have *honest* users who *want* to protect against untrusted applications and user error. While protecting against malicious users is also important, current approaches for addressing these privacy concerns do not work well *even for honest users.* In this paper we thus assume that users are honest, but that they may run untrusted applications.

Sensitive locations like locker rooms and bars commonly handle these concerns today by posting explicit policies that prohibit recording or the presence of recording devices (including Google Glass [60]), as in Figure 5.1. This approach, however, is hard to enforce and does little to protect a user from untrusted applications or user error. Users must notice the sign, then remember to turn off their device.

A new access control challenge. A natural way to address these privacy concerns is with application permissions in the operating system. However, continuous sensing and natural user input pose new challenges to access control design. Today, platforms like Android, iOS, and Windows 8 deny untrusted applications default access to sensitive resources like the camera and GPS. To determine which permissions to grant, these OSes put the user in the loop: with manifests at application installation time (Android, Windows 8) or prompts at the time of sensitive data access (iOS).

Previous work [50, 138], however, has shown that these permission models are flawed. Manifests are out of context with applications' use of sensitive data, making it hard for users to understand what permissions applications need and why. Prompts are disruptive and cause "prompt fatigue," conditioning users to simply click yes.

User-driven access control (Chapter 3) addresses these flaws by coupling permission granting with user actions within an application (e.g., clicking a special embedded camera button). Unfortunately, this approach is not well-suited for continuous sensing because it relies on explicit user interactions with the device. By contrast, continuous sensing applications are, almost by definition, designed to automatically "do things for you" without any such explicit actions.

Further, for applications with natural user input, like gesture or voice, the input method itself relies on the camera or microphone being always accessible. In these settings, permission granting models that allow or deny access to entire sensor streams are too coarse-grained.

The fundamental new challenge in permission system design for continuous sensing is thus enabling *fine-grained, automatic* permission granting. For example, how should the OS determine which objects in a video frame are accessible to an application? We pursued multiple unsuccessful attempts to design access control for continuous sensing, which we discuss in Section 5.4.1. From them, we learned that access control decisions should depend on objects and people around the user, and that these objects should specify their own access policies in a distributed, context-sensitive way.

World-driven access control. Our solution is *world-driven access control.* Using this approach, objects, places, and people would present *passports* to the operating system. A passport specifies how to handle access control decisions about an object, along with, optionally, code stating how the object should be recognized. For example, in Figure 5.1, the locker room might present a passport suggesting that video or audio recording is prohibited. Passports provide a *distributed, context-sensitive* approach to access control.

In our design (Section 5.4), a trusted *policy module* in the OS detects passports, extracts policies, and applies those policies dynamically to control applications' access to sensor data. Our design protects the user from untrusted applications while relieving the user of explicit permission management. While users can override policies communicated by passports, applications cannot.

Passports are intended to help users avoid accidentally sharing or allowing applications to access sensitive data. For example, a workplace can publish a passport stating that whiteboards are sensitive, helping the user avoid recording (and later accidentally sharing on social media) photos of confidential information on the whiteboard. In the locker room, a "no-record" policy helps the user avoid accidentally allowing an untrusted application to access the video feed of herself undressing in the mirror.

Passports can also help users respect others' wishes without requiring onerous manual configuration. At the AdaCamp conference, for example, attendees wear red lanyards to opt out of photography [4]. A world-driven access control policy can tell the policy module to remove those attendees from video streams and photos before applications see them. The user does not need to manually check lanyards or remove the device entirely. Our approach allows dynamically changing application permissions based on *context*, such as being at a conference, without explicit user actions.

Making world-driven access control work requires overcoming multiple challenges. First, there are many different *policy communication mechanisms*, ranging from QR codes and Bluetooth to object recognition, each with different tradeoffs. Second, recognizing passports and computing policy decisions induces *latency* for applications. Third, policy decisions may have *false positives and false negatives.* Finally, our approach creates a new problem of *policy authenticity* as adversaries may attempt to move, modify, or remove markers that communicate policies. We describe these and other challenges, as well as our approaches for addressing them, in the context of our implementation in Section 5.6 and our evaluation in Section 5.7.

Contributions. We introduce *world-driven access control*, whereby application access to sensor data depends on policies specified by real-world objects. Our approach allows the system to automatically manage application permissions without explicit user interaction, and supports permissions at the granularity of real-world objects rather than complete sensor streams. We contribute:

1. *World-driven access control*, a permission model for continuous sensing that allows real-world objects to communicate policies using special *passports*. Our design enables a distributed, context-sensitive approach for objects to control how sensitive data about them is accessed and for the system to validate the authenticity of these policies.
2. An extensible system design and implementation in which a trusted *policy module* protects users from untrusted applications without requiring the user to explicitly manage permissions. We evaluate our prototype in five different settings (Section 5.7) with representative policies from prior work and real-world policies. Our design's modularity allows us to implement each policy in under 150 lines of C# code.
3. Empowering objects with the ability to influence access control decisions on users' devices introduces numerous challenges. We crystallize these challenges and explore methods for addressing them. For example, we introduce techniques for mitigating latency and accuracy challenges in detecting and enforcing policies (Sections 5.4-5.7), such as by combining multiple means of communicating a policy.

In summary, world-driven access control is intended to to relieve the user's permission management burden while preserving functionality and protecting privacy in emerging continuous sensing applications. This work presents a new design point in the space of access control solutions for continuous sensing applications. We believe that the foundations laid herein will facilitate further work in the field.

Previous work in this area focused on bystander privacy (e.g., visual or electronic opt-outs [21, 63, 145]), or is specific to one type of object alone. We discuss related work in detail in Section 5.9. Finally, while our focus is on continuous sensing, our policies can also apply to discrete sensing applications, e.g., a cell phone camera taking a photo, as well as to output permissions, e.g., permission for a camera to flash or a phone to ring. We reflect on challenges and discuss other extensions in Section 5.8, then conclude in Section 5.10.

5.2 Goals and Threat Model

We consider fine-grained access control for sensor data on platforms where multiple isolated applications desire continuous access to system sensors, such as camera, microphone, and GPS. In our model, the system is trustworthy and uncompromised, but applications are untrusted.

Goals. Our goal is to *help honest users manage applications' permissions.* A user may do so to (1) *protect his/her own privacy* by minimizing exposure of sensor information to untrusted applications, and/or (2) *respect bystanders' privacy wishes.* We seek to help the user achieve these goals with minimal burden; users should not need to continuously manage permissions for each long-running application.

Constraints. We constrain our model in three ways:

1. We consider only access control policies that are applied at data collection time, not at data distribution time. These policies affect whether or not an application may receive a data item — such as a camera frame or audio event — but do not provide additional data flow guarantees after an application has already accessed the data.
2. Policies apply to applications, not to the system itself. For example, if a policy restricts camera access, the (trusted) system still receives camera data but prevents it from reaching applications. We observe that any system designed to apply policies embedded in real-world sensor data must, by definition, be able to receive and process sensor inputs.
3. Users can always use another device that does not run our system, and hence can, if they desire, violate real-world policies with non-compliant devices. Our solution therefore does *not* force user or device compliance and, indeed, explicitly allows users to override access control policies. We consider techniques for verifying device compliance, which have been studied elsewhere [94], out of scope.

Novel threat model. Our approach empowers real-world objects with the ability to broadcast data collection policies. Unlike conventional approaches to access control, like manifests and prompts, we therefore transfer the primary policy controls away from the user and onto objects in the real world. Thus, in addition to untrusted applications, we must consider the threat of *adversarially-influenced real-world objects*. For example, we must consider malicious policies designed to prevent recording (e.g., criminals might appreciate the ability to turn off all nearby cameras) or override policies that prohibit recording (e.g., someone seeking to embarrass users in the bathroom by causing them to accidentally record). These threats guide several design decisions, which we will return to later, like the ability for users to override policy suggestions and our design of passports.

5.3 Real-World Policies

We summarize a sample set of real-world locations with sensor privacy policies (Table 5.1) that would benefit from world-driven access control. These scenarios motivate the incentive of establishments to install world-driven policies.

The first two settings are bars that request patrons refrain from taking recordings of others without permission. For example, the 5 Point Cafe recently banned Google's Glass device due to privacy concerns [163] and other Glass bans have followed [60]. Similarly, the "local gym" and "Goodwill" settings both prohibit recording devices near dressing rooms. All of these places could specify an explicit "no pictures" policy for a world-driven access control device.

The next four settings are conventions that set explicit policies around photography [4], generally requiring asking before taking a photo or prohibiting recording in certain times or locations (e.g., during a conference session). One policy is more fine-grained, based on explicit communication from attendees using colored lanyards (where a red lanyard means "no photos," a yellow lanyard means "ask first," and a green lanyard means "photos OK"). These colors could serve as the foundation for a world-driven access control policy. As suggested in prior work, such individual policies could also be communicated using a marker on a shirt [145] or via a communication from the bystander's phone (e.g., [21, 63]).

Finally, Google's Street View product, which is powered by cars that regularly drive

Place	*Policy*
Mercury	Ask before photo.
Mercury	Avoid people in background of photo.
5Point Cafe	No photos, no Google Glass.
Goodwill	No recording w/in 15 ft. of dressing room.
Local Gym	No cell phone in locker room.
AdaCamp SF	Lanyard color indicates photo preference.
Open Source Bridge	Provide notice before taking photo.
Sirens	Ask before photo.
Sirens	No photos during sessions.
WisCon	Ask before photo.
Street View	Blur sensitive areas, no photos over fences.

Table 5.1: **Existing Real-World Policies.** Today, multiple places request that people follow photo and video policies. We report a sample set of places, along with the policy they request. These policies can be captured by our framework.

through the world and capture panoramic views, has raised significant privacy concerns. Switzerland only recently allowed Google to collect Street View data, but it required Google to blur sensitive areas and refrain from taking pictures across fences and garden hedges [125]. These policies could also be enforced by world-driven access control, based on the locations of sensitive areas or presence of fences.

While today's real-world policies tend to focus on the privacy of bystanders, always-on devices like Google Glass will also raise privacy concerns for users. For example, the user survey that we describe in Section 5.7.1 suggests that many users do not wish to be recorded by their own wearable devices in sensitive locations (e.g., at home, in the bathroom).

5.4 World-Driven Access Control Design

World-driven access control is based on three principles derived from our initial attempts to build an access control mechanism for continuous sensing, which we summarize. We then describe our solution, key challenges we encountered, and how our design addresses these challenges. Figure 5.2 shows world-driven access control in action, and Figure 5.3 shows a block diagram of our system.

5.4.1 Principles from Initial Approaches

Object-driven policies. In user-driven access control (Chapter 3), applications are granted access to sensitive resources as a result of explicit in-application user interactions with special UI elements (e.g., clicking on an embedded camera button). This technique effectively manages application permissions without the drawbacks of prompts or install-time manifests, and so we attempted to adapt it to continuous sensing.

In some cases, user-driven access control worked well — photos and video chat on Google Glass, for example, are triggered by explicit user actions like winking or using the touchpad. However, many promising continuous sensing applications do not involve explicit user input. For example, the WordLens translation application is *object-driven*, rather than user-driven: it reacts to all words it “sees” and translates them without explicit user input. In future systems, such applications that “do things for you” may run continuously for long periods of time; they are uniquely enabled by continuous sensing and raise the greatest privacy risks. Attempting to inject user actions into such applications — such as allowing users to use a drag-and-drop gesture to grant applications one-time access to information about real-world objects — felt disruptive and artificial.

From this, we learned that access control decisions cannot depend (only) on user actions but should instead be object-driven: real-world objects around the user must help determine access control policies.

Distributed organization. The next problem was how to map from objects to access control policies. We first attempted to build a taxonomy of which objects are sensitive (e.g., whiteboards) or not sensitive (e.g., chairs). Unfortunately, a static taxonomy is not rich enough to capture the complexities of real-world objects — for example, not all whiteboards contain sensitive content.

We then attempted to define a hierarchical naming scheme for real-world objects. For example, we might name an object by referring to the room it is in, then the building, and finally the building’s owner. We hoped to build a system analogous to the Web’s Domain Name System, where objects could be recognized in the real world, mapped to names, and finally mapped to policies. Unfortunately, because objects can move from one location to another, policies may not nest hierarchically, and hierarchies not based on location quickly become complex.

Though there may be value in further attempting to taxonomize the world and its moving objects, we chose to pursue an approach in which policy decisions are not centralized. Instead, our design's access control decisions are *distributed*: each object is able to set its own policy and communicate it to devices.

Context sensitivity. Finally, we faced the challenge of specifying an object's policy. A first approach here is to have static "all-or-nothing" policies. For example, a person or a whiteboard could have a policy saying to never include it in a picture, and certain sensitive words might always mark a conversation as not for recording.

While static policies cover many scenarios, others are more complex. For example, AdaCamp attendees can opt out of photos by wearing a red lanyard [4]. At the conference, the lanyard communicates a policy, but outside it does not. Policies may also depend on applications (e.g., only company-approved apps may record a whiteboard). From this, we learned that policies must be *context-sensitive* and arbitrarily complex. Thus, in addition to static policies, we support policies that are (appropriately sandboxed) executable code objects, allowing them to evaluate all necessary context.

5.4.2 Background Technology: Recognizers & Events

A key enabler for world-driven access control is the *recognizer* operating system abstraction [75]. A recognizer is an OS component that processes a sensor stream, such as video or audio, to "recognize" objects or other triggers. Upon recognition, the recognizer creates an *event*, which is dispatched by the OS to all registered and authorized applications. These events expose higher-level objects to applications, such as a recognized skeleton, face, or QR code. Applications may also register for lower-level events, like RGB/depth frames or location updates.

The recognizer abstraction restricts applications' access to raw sensor data, limiting the sensitive information an application receives with each event. Further, this event-driven

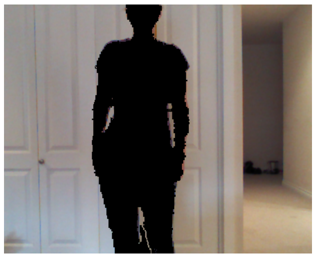


Figure 5.2: **World-Driven Access Control in Action.** World-driven access control helps a user manage applications' permission to both protect his or her own privacy and to honor a bystander's privacy wishes. Here, a person in view of a camera asks not to be recorded; our prototype detects and applies the policy, allowing applications to see only the modified image with the person removed. In this case, the person wears a QR code to communicate her policy, and a depth camera is used to find the person, but a variety of other methods can be used.

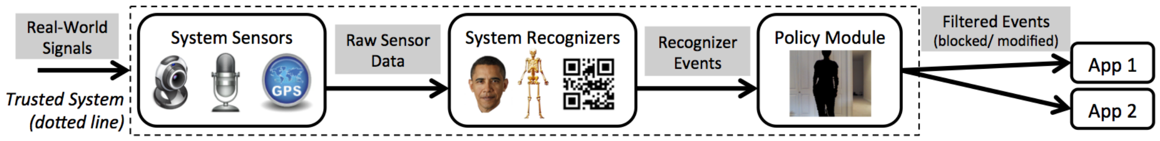


Figure 5.3: **System Overview.** We build on the recognizer abstraction [75], in which applications subscribe to high-level events generated by object recognition algorithms. Our contribution is the policy module (Section 5.4.4), which sees all events, detects policies, and blocks or modifies events accordingly before releasing them to untrusted applications.

model allows application developers to write code agnostic to the application's current permissions [75]. For example, if an application does not receive camera events, it cannot distinguish whether the hardware camera is turned off or whether it is currently not authorized to receive camera events.

Previous work on recognizers, however, did not study how to determine which permissions applications should have [75]. While the authors discuss visualization techniques to inform users what information is shared, the burden is on users to manage permissions through prompts or manifests. World-driven access control removes this burden by dynamically adjusting permissions based on world-expressed policies.

5.4.3 Policies and Policy Communication

We now dive more deeply into our design. We seek a general, extensible framework for communicating policies from real-world objects. Our prototype focuses primarily on policies *communicated explicitly* by the environment (e.g., by QR code or ultrasound), not inferred by the system. However, our system can be extended to support implicit or inferred policies as computer vision and other techniques improve. For example, the recent PlaceAvoider approach [160] — which can recognize privacy-sensitive locations with training — can be integrated into our system.

Designed to be a broad framework, world-driven access control can support a wide range of policy communication mechanisms on a single platform, including QR codes or other visual markers, ultrasound, audio (e.g., embedded in music), location, Bluetooth, or other wireless technologies. Additionally, world-driven access control supports both policies that completely block events (e.g., block RGB frames in the bathroom) and that selectively modify events (e.g., remove bystanders from RGB frames). Section 5.3 summarized existing real-world policies (e.g., recording bans) that can be supported by this approach and which motivate the incentives of establishments to deploy world-driven policies.

A world-driven policy specifies (1) when and for how long it should be active, (2) to which real-world objects or locations it applies, (3) how the system should enforce it, and (4) to which applications it applies, if applicable. Elaborating on the latter, policies can be application-specific, such as by whitelisting known trusted applications. For example, a corporate building's policy might block audio input for any application not signed with the company's private key.

While conceptually simple, there are numerous challenges for communicating policies in practice. In our design, we aim to quickly recognize a policy, quickly recognize the specific objects or locations to which it applies, and accurately enforce it. A challenge in achieving these goals is that policy communication technologies differ in range, accuracy, information density, and the ability to locate specific real-world objects. For example, Bluetooth or WiFi can be detected accurately across a wide area but cannot easily refer to specific objects (without relying on computer vision or other object detection techniques). By contrast, QR codes can help the system locate objects relative to their own location (e.g., the person to whom the QR code is attached), but their range and detection accuracy is limited (Section 5.7). Similarly, a QR code can communicate thousands of bytes of information, while a specific color (e.g., a colored shirt or lanyard [4, 145]) can communicate much less.

We begin to tackle these challenges with several approaches, and return to additional challenges (like recognition latency and policy authenticity) in later sections:

Extending policy range. The range of a technology should not limit the range of a policy. We thus allow a policy to specify whether it is active (1) while the policy signal itself is in range, (2) for some specified time or location, or (3) until an explicit "end policy" signal is sensed. For example, a WiFi-based policy for a building might use the first option, and a QR-code-based policy for a bathroom might use the third option (with "start" and "end" policy QR codes on either side of the bathroom door). There are challenges even with these approaches; e.g., the "end policy" signal may be missed. To overcome missed signals, the QR-code-based policy can fall back to a timeout (perhaps after first notifying the user) to avoid indefinitely applying the policy.

Increasing information density. Policies need not be specified entirely in a real-world signal. Instead, that signal can contain a pointer (such as a URL) to a more detailed remote policy. Based in part on this idea, we introduce passports as a method for dynamically extending the system with new policy code in Section 5.5.

Combining technologies. Observing that different communications methods have different tradeoffs, we propose the use of *hybrid techniques*: allowing multiple mechanisms to work together to express a policy. For example, to remove bystanders wearing a specific color from RGB frames, a wide-range high-density technology like Bluetooth can bootstrap the policy by informing the system of the opt-out color; the color helps locate the area to remove from a frame.

5.4.4 Policy Detection and Enforcement

There is considerable diversity in the types of policies objects may wish to convey, and hence our solution must be able to accommodate such diversity. As surfaced in this and later sections, not all policies are trivial to handle.

World-driven policies are enforced by a trusted policy module on the platform (Figure 5.3). The policy module subscribes to all (or many) of the system recognizers. Any of these recognizers may produce events that carry policies (e.g., detected QR codes, WiFi access points, or Bluetooth signals). The policy module filters events before they are delivered to applications. The result of event filtering may be to block an event or to selectively modify it (e.g., remove a person from an RGB frame).

In more detail, the policy module keeps the following state: (1) default permissions for each application (e.g., from a manifest or other user-set defaults), specifying whether it may receive events from each recognizer, (2) currently active policies, and (3) a buffer of events from each recognizer.

Before dispatching a recognizer event to an application, the policy module consults the default permission map and all active policies. Depending on the result, the policy module may block or modify the event.

Event modification is complicated by the fact that it may rely on information in multiple events. For example, if someone wearing a QR code should be removed from an RGB frame, the modification relies on (1) the person event, to isolate the pixels corresponding to a person, and (2) the QR code event, to pinpoint the person nearest the QR code's bounding box (to whom the policy applies).

The policy module thus uses the buffer of recognizer events to identify those needed to enforce a policy. Because corresponding events (those with the same frame number or similar timestamps) may arrive at different times from different recognizers, the policy module faces a tradeoff between policy accuracy and event dispatch performance. For example, consider a QR code that begins an RGB blocking policy. If the QR code recognizer is slow and the policy module does not wait for it, an RGB frame containing the QR code may be mistakenly dispatched before the policy is detected.

```
public interface IPolicy {
 string PolicyName();
 void Init();
 void ApplyPolicyToEvent(
  RecognizerEvent inEv, EventBuffer prevEvents,
  out RecognizerEvent outEv, out bool modified);
 void Destroy();         }
```

Figure 5.4: **Policy Interface.** Policies implement this interface. The policy module calls `ApplyPolicyToEvent()` for each active policy on a new event for an application. The policy implementation modifies or blocks the event, and the policy module forwards the result to the application.

Thus, for maximum policy accuracy, the policy module should wait to dispatch events until all other events on which a policy may depend have arrived. However, waiting too long to dispatch an event may be unacceptable, as in an augmented reality system providing real-time feedback in a heads-up display. We discuss our implementation choice, which favors performance, in Section 5.6.

5.4.5 Policy Interface

Each world-expressed policy has a fixed interface (Figure 5.4), which allows policy writers to extend the system's policy module without dictating implementation. We expect that policies are typically short (those we describe in Sections 5.6 and 5.7 all require under 150 lines of C#) and rely on events from existing object recognizers, so policy writers need not implement new recognizers.

The key method is `ApplyPolicyToEvent`, which is called once for each event dispatched to each application. This method takes as input the new event and a buffer of previous events (up to some bound), as well as metadata such as the application in question and the system's configuration (e.g., current permissions). The policy implementation uses this information to decide whether and how to modify or block the event for this application. The resulting event is returned, with a flag indicating whether it was modified.

The resulting event is then passed to the next policy, which may further modify or block it. In this way, policies can be composed. Although later modifications are layered atop earlier ones, each policy also receives *unmodified* events in the event buffer used to make and enforce policy decisions. Thus, poorly written or malicious policies cannot confuse the decisions of other policies.

Policy code isolation. Policy code can *only* use this restricted API to obtain and modify sensor events. Policies must be written in managed code and may not invoke native code, call OS methods, or access the file system, network, or GPU. (Isolating managed code in .NET is common with AppDomains [111].) Our restrictions protect the user and the OS from malicious policy code, but they do impose a performance penalty and limit policies to making local decisions. Our evaluation shows that we can still express many realistic policies efficiently.

Policy conflicts. In the case of policy conflicts, the system must determine an effective global policy. If multiple world-driven policies conflict, the system takes the most restrictive intersection of these policies. In the case of a conflict due to a user's policy or explicit action, recall that we cannot force users or their devices to comply with world-driven policies, since users can always use hardware not running our system. The system can thus let the user override the policy (e.g., by incorporating access control gadgets [138] to verify that the action genuinely came from the user, not from an application) and/or use a confirmation dialog to ensure that the violation of the world-driven policy is intentional, not accidental, on the user's part. This approach also gives users the ability to override malicious or questionable policies (e.g., policies that block too much or too little).

5.5 Passports: Policy Authenticity and Runtime Extensions

In this section, we simultaneously address two issues: policy authenticity and system extensibility. First, an attacker may attempt to forge, move, or remove an explicit world-driven policy, requiring a method to verify policy authenticity. Second, new policy communication technologies or computer vision approaches may become available, and our system should be easily extensible.

To address both issues, we introduce a new kind of digital certificate called a *passport*.

Inspired by real-world travel passports, our policy passport is designed to support policy authenticity by verifying that a passport (1) is issued by a trusted entity, and (2) is intended to apply to the object or location where it is observed. To support extensibility, a passport may additionally contain sandboxed object recognition and/or policy code to dynamically extend the system.

5.5.1 On the Need for Policy Authenticity

We elaborate here on the need to address attackers who might wish to make unauthorized modifications to existing policies. Attacker modifications may make policies less restrictive (e.g., to trick employees of a company into accidentally taking and distributing photos of confidential information in conference rooms) or more restrictive (e.g., to prevent the cameras of surrounding users from recording the attacker breaking a law). In particular, attackers attempting to change policies may employ the following methods:

1. Creating a forged policy.
2. Moving a legitimate policy from one object to another (effectively forging a policy on the second object).
3. Removing an existing policy.

The difficulty of mounting such attacks depends on the policy communication technology. For example, it may be easy for an attacker to replace, move, or remove a QR code, but more difficult to do so for an ultrasound or WiFi signal. Thus, these risks should be taken into account by object or location owners deploying explicit policies.

The threat of policy inauthenticity also varies by object. For example, it is likely more difficult for an attacker to forge a policy (like a QR code) affixed to a person than one affixed to a wall. Thus, for certain types of objects, such as humans, the system may trust policies (e.g., colored lanyards as opt-outs [4]) without requiring additional verification, under the assumption that it is difficult to physically manipulate them.

5.5.2 Signing Passports

The contents and origin of passports must be cryptographically verifiable. There are numerous approaches for such verifiability, ranging from crowd-sourced approaches like Perspectives [174] and Convergence [101] to social-based approaches like PGP [182]. While the certificate authority (CA) model for the Web has known limitations [27], because of its (current) wide acceptance, we choose to build on the CA model for world-driven access control. However, we stress that the underlying mechanism is interchangeable.

Building on the CA model, we introduce a policy authority (PA), which is trusted to (1) verify that the entity requesting a passport for an object or location is the real owner (i.e., has the authority to provide the policy), and (2) sign and issue the requested passport. Thus, object or location owners wishing to post policies must submit them to be signed by one of these trusted authorities. For this approach to be effective, it must be difficult for an attacker to obtain a legitimately signed policy for an object or location for which he or she cannot demonstrate ownership.

5.5.3 Describing Objects via Passports

In addition to containing a policy specification, each signed passport contains a description of the associated object or location. For a fixed location (e.g., on a corporate campus), the description may specify GPS coordinates bounding the targeted area. For a mobile object, the description should be as uniquely-identifying as possible (e.g., containing the license plate of the associated car). Thus, after verifying a passport's signature, the system also verifies that the enclosed description matches the associated object or location.

Having a description in the passport helps prevent an attacker from moving a legitimately signed passport from one object or location to another. Preventing such an attack may be difficult to achieve in general, as it depends on the false positive rates of the object recognition algorithm and what is actually captured by the device's sensors (e.g., the car's license plate may not be in view). The key point is that passports give object owners control over the description, so an object owner can pick the best available description.

We observe that object descriptions may create privacy risks for objects if not designed carefully. We return to the tension between description clarity and privacy in Section 5.8.

We further support *active object descriptions* contained in the passport. Rather than being simply static (e.g., “car with license plate 123456”), the object description may consist of code (or of a pointer to code) that directly extends the system’s recognizers (e.g., with a car recognizer). Beyond aiding policy verification, active descriptions can extend the system’s basic object recognition abilities without requiring changes to the system itself. Certificates including code and policy are similar in spirit to the permission objects in .NET Code Access Security [115], the delegation authorization code in Active Certificates [20], the self-describing packets in Active Networks [162], or objects in Active DHTs [53], but we are not aware of previous approaches that dynamically add object recognition algorithms. In our initial design, the active object descriptions are trusted by virtue of the PA’s signature, and could be further sandboxed to limit their capabilities (as in Active DHTs [53]).

If the description contained in the passport does not match the associated object, the passport is ignored — i.e., the system applies a sensible default. For a real-world travel passport, the default is to deny access; in our case, requiring explicit allow-access policies on all objects would likely require an unacceptable infrastructure and adoption overhead. Thus, in practice, the system’s default may depend on the current context (e.g., a public or private setting), the type of object, or the type of recognizer. For example, a default may deny face recognition but allow QR code events.

We believe the idea of active object descriptions is of interest independent from world-driven policies. By allowing real-world objects to specify their own descriptions, the system’s object recognition capabilities can be easily extended in a distributed, bottom-up manner.

5.5.4 Monitoring Passports

Passport signatures and objects descriptions help prevent attackers from forging or moving passports. We must also consider the threat of removing passports entirely. Again, because a deny-access default may create high adoption overhead, a missing passport allows access in the common case.

As an initial approach, we suggest mitigating this threat by monitoring passports. In particular, a policy owner can monitor a passport directly — by installing a device to actively monitor the policy, e.g., to ensure that the expected QR code or ultrasound signal is still present — or can include in the policy itself a request that devices occasionally report back policy observations. While an attacker may also try to manipulate the monitoring device, its presence raises the bar for an attack: disabling the monitor will alert the policy owner, and fooling it while simultaneously removing the policy from the environment may be difficult for some policy communication technologies (e.g., WiFi).

Stepping back, we have explored the problem space for policy passports. The proposed designs are an initial approach, and we welcome future work on alternative approaches supporting our core concept that *real-world objects can specify their own policies and prove their authenticity.*

5.6 Implementation

While world-driven access control may intuitively appear straightforward, it is not obvious that it can actually work in practice. We explore the challenges — and methods for overcoming them — in more detail with our implementation and evaluation. Our prototype runs on a Windows laptop or tablet and relies on sensors including the Kinect's RGB/depth cameras and its microphone, the built-in WiFi adapter, and a Bluetooth low energy (BLE) sensor. Though this platform is more powerful than some early-generation continuous sensing devices, we expect these devices to improve, such as with specialized computer vision hardware [29, 127]. Some continuous sensing platforms — such as Xbox with Kinect — are already powerful enough today.

Sensor input is processed by recognizers [75], many of which simply wrap the raw sensor data. We also implemented a QR code recognizer using an existing barcode library [1], a speech keyword recognizer using the Microsoft Speech Platform, a WiFi access point recognizer, and one that computes the dominant audio frequency.

5.6.1 Policy Module

As described in Section 5.4.4, the policy module subscribes to events from all of the system's recognizers and uses them to detect and enforce policies. Before dispatching an event to an application, the system calls the policy module's `FilterEvent()` method, which blocks or modifies the event based on currently active policies. To modify an event, it uses recently buffered events from other recognizers if necessary.

Recall from Section 5.4.4 the policy accuracy versus performance tradeoff: policies can be detected and applied more accurately if all relevant events have arrived, but waiting too long impacts event dispatch latency. In our implementation we favor performance: to avoid delays in event dispatch, we do not match up events precisely using frame numbers or timestamps, but simply use the most recent events, at the possible expense of policy accuracy. Alternatively, this choice could be made per application or by user preference. We quantify the effects of this tradeoff in Section 5.7.

5.6.2 Passports

In addition to static policies (e.g., a simple QR code that activates a pre-installed "block RGB events" policy), our prototype supports passports by transforming certain recognizer events into passport lookups. The system appends the text in a QR code or a Bluetooth MAC address to the base URL at `www.wdac-passport-authority.com/passportlookup/`. If a corresponding passport exists, the server provides its URL. The passport contains a policy DLL that implements our policy interface (Figure 5.4). The system downloads, verifies, and installs the new policy (if not previously installed).

The process of loading a passport could be optimized by caching server responses. If the system cannot make a network connection, it misses the passport; a future implementation could buffer the network request until connectivity is available, in case the policy is still applicable then. Note that the system does not need a network connection to parse static (non-passport) policies; passports communicated via certain communication technologies (e.g., Bluetooth) could also encode their policy code directly rather than pointing to a server. As noted in Section 5.5, code in a passport could be signed by a policy authority and sandboxed. Since signature and sandboxing approaches have been well studied elsewhere, we do not include them in our prototype but rather focus on world-driven access control functionality.

5.6.3 Prototype Policies

Block RGB in sensitive area. We implemented several policies blocking RGB events in sensitive areas, based on QR codes, BLE, WiFi, and audio. For example, a QR code on a bathroom door can specify a policy that blocks RGB until the corresponding "end policy" QR code is detected. We can similarly use a BLE emitter to specify such a policy; BLE has a range of roughly 50 meters, making it suitable for detecting a sensitive area like a bathroom. Similarly, we use the access point (AP) name of our office's WiFi network to trigger a policy that blocks RGB events. (Future APs might broadcast more complete, authenticated policies.)

We also use our audio frequency recognizer to block RGB events: if a trigger frequency is heard three 32 ms timeslots in a row, the corresponding policy is engaged (and disengaged when the trigger has not been heard for three consecutive timeslots). In our experiments, we used a frequency of 900 Hz; in a real setting, ultrasound may be preferable. Ultrasound is already used in real settings to communicate with smartphones [165]. More complex audio communication is also possible, e.g., encoding a policy in the signal [133].

Remove person from RGB. We support several policies to remove bystanders from RGB frames. First, we support bystanders wearing opt-out QR codes — though our implementation could easily support an opt-in instead — and filter RGB events to remove the pixels associated with the person nearest such a QR code's bounding box. To do so, we use the per-pixel `playerIndex` feature of Kinect depth frames. Figure 5.2 shows a modified RGB frame based on this policy.

As a simpler, more efficient policy to remove people from RGB frames, and inspired by practices at AdaCamp, we also implemented a color-based policy, using a certain shirt color to indicate an opt-out. This policy (1) detects a 10x10 pixel square of the target color, (2) calculates its average depth, and (3) removes RGB pixels near that depth. Thus, this policy applies to anything displaying the target color, and it avoids the latency of the Kinect's player detection. Using a hybrid approach, we also implemented a BLE policy to bootstrap the color policy (discussed more in Section 5.7).

Block sensitive audio. We implemented a policy that blocks sensitive audio based on our speech keyword recognizer. We use a hard-coded grammar that includes words that may signal a sensitive conversation, such as project codenames like “Natal,” as well as explicit user commands such as “stop audio.” The policy blocks audio events once such a keyword is detected, and resumes audio events upon command (i.e., when the user explicitly says “begin audio”).

5.7 Evaluation

We evaluate along two axes:

1. *Policy Expressiveness and Implementation Effort.* We show that our architecture can incorporate a wide variety of policies desired by users and proposed in prior work with low developer effort (Section 5.7.1).
2. *Policy Effectiveness.* We explore the effectiveness of policy communication technologies and world-driven policies implemented in our prototype (Section 5.7.2).

5.7.1 Policy Expressiveness & Implementation Effort

User Survey

We first conducted an online survey to study attitudes toward continuous video sensing. This study adhered to human subjects research requirements of our institution. The results confirm our intuition: users do desire privacy from continuous sensing applications. We conclude that our design addresses a threat that causes concern, and that our policy abstraction is expressive enough to be useful.

In more detail, we conducted an online survey of 207 anonymous participants recruited from uSamp, a professional survey service. We aimed to: (1) determine if people are familiar with wearable cameras, which can be a particularly privacy-sensitive form of continuous sensing, (2) collect privacy policies that people might want for continuous sensing systems, and (3) get feedback about specific threats, including the threat of an untrusted application reading video data.

We showed participants two short videos about wearable cameras. The first video was a demonstration of the GoPro wearable camera at the 2011 Consumer Electronics Show [22], created by an independent news site `butterscotch.com`. The second video was a review of Google Glass by the news site CNET.com [28]. We then asked participants about their familiarity with wearable cameras, clarifying that we did not mean to include standard cell phones. Subsequently, we asked participants to consider which parts of the day they would or would not want to record using their own wearable cameras and why. Finally, we asked participants to rate on an absolute Likert scale how concerned they were about four possible threats raised by wearable cameras.

We filtered out participants who failed to finish the survey. While we could not ensure participants paid attention to our videos, we also filtered out participants who answered the survey too quickly to have finished watching the videos. After filtering, we had 131 total responses. Respondents were 35% male, 65% female, with 9% 18-24, 29% 25-34, 25% 35-44, 17% 45-54, 12% 55-64, and 5% above 65 (percentages do not sum to 100% due to rounding). We learn:

Wearables are known but still niche. Over half (58%) of participants were familiar with wearable cameras before watching the videos in our survey. Only 6% had used a wearable camera themselves, and only 11% had been recorded by one, but 20% knew someone else who did use a wearable camera. Free-text responses mentioned sports, such as skydiving, piloting, and skiing, as use cases for wearable cameras. One respondent mentioned *I don't know them personally but my police officer friend uses one at work*, as a work-related use case. We see this result as evidence that the time is right to explore privacy-preserving mechanisms for continuous sensing systems, before they become ubiquitous.

Time and location are important privacy policy factors. We asked respondents: *Imagine your entire day being recorded by YOUR OWN wearable camera. Which parts of your day are you comfortable recording? Which parts of your day are you not comfortable recording? For the purposes of this question, please do not include unmodified cell phones.* From these free-text answers, we then manually determined if the answer depended on time or location. For example, the answer "When I am in the bathroom and getting dressed in the morning" depends on time or location, but "anything goes" does not.

We found that 45% of respondents gave answers depending on time or location. Bathrooms and homes were common places where respondents did not feel comfortable recording, suggesting that our system should support policies that identify specific places and modify events passed to untrusted applications accordingly.

A respondent from an earlier pilot survey commented: *[I] could talk about unrealistic things like a camera's ability to somehow detect people that don't want to be recorded (without using a phone app) or to allow commands from subjects being recorded to command the device to cease and erase the recording, but that's largely unrealistic.* World-driven access control aims to make such policies realistic.

Multiple threats are important, including untrusted applications. We asked participants to consider four distinct threats that would cause them to be uncomfortable recording video on their own wearable device. For each threat, we asked participants to state how important the threat is on a 7-point Likert scale. Participants were not asked to relatively rank threats, i.e., they did not state which threat was "most important."

For the threat "An application on my phone or device might see the video", 44% of respondents rated it as 7, or highly important. (We consider only the highest importance score to estimate a lower bound of users truly concerned about this threat.) This is the threat we address in our design. The other three threats and the percentage rating as highly important were "The device manufacturer might look at the video" (39%), "I might accidentally share the video" (48%), and "I might give my device to someone else with the video on it" (45%). Respondents also shared additional threats, including concerns about access by the U.S. National Security Agency and the threat of an external hacker compromising the device.

Our design addresses some, but not all, of these threats. We directly address untrusted applications. We also partially address accidental sharing, because our policies prevent sensitive data from reaching an application that may later share it. Future work may extend world-driven access control to address additional threats.

Comparison to in-depth studies. While our surveys reached over 100 people, we could not perform in-depth follow-up with respondents. Others have studied people's privacy

Name	*Detection*	*Enforcement*	*Lines of Code (C#)*
Respectful Cameras [145]	Visual indicator	Blur faces	84 (QR code) – 104 (color)
Brassil [21]	Bluetooth	Blur faces	69 (remove person)
TagMeNot [23]	QR Code	Blur faces	84 (remove person)
Iceberg Systems [84]	Ultrasound	Disable camera	35 (audible sound)
Parachuri et al. [128]	Visual indicator	Remove person	84

Table 5.2: **Policies in Prior Work.** Our architecture generalizes policies from prior work; this table details those we implemented.

concerns with respect to in-home sensors and video surveillance in more depth (e.g., [24, 25, 102]). While these studies did not, like our, ask explicitly about untrusted applications, they give valuable insight into the threats seen by people who are living with continuous sensing and how those might be addressed with world-driven access control.

Policies in Prior Work

We now validate that our policy interface (Figure 5.4) is expressive enough to capture realistic policies. We implemented a representatives set of policies proposed in prior work relevant to continuous sensing [21, 23, 84, 128, 145], summarized in Table 5.2. In some cases, we implemented a slightly different enforcement than the one specified in prior work (e.g., removing people from the frame instead of blurring faces), and we note these differences where applicable. Each policy could be implemented in our prototype with only a modest number of lines of C# code (between 35 and 104, measured using Visual Studio's Code Metrics), indicating that our architecture supports new and diverse policies with modest additional developer effort.

Other policies exist in prior work, such as a policy to encrypt recordings with a shared key derived with a distributed prototol [63]. We determined that our system could support this policy, but for our purposes here we chose not to spend effort reimplementing its protocol. Our prototype can also handle existing real-world policies (Appendix 5.3).

We conclude that our architecture's policy abstraction is sufficiently expressive and that policies can be implemented with modest developer effort.

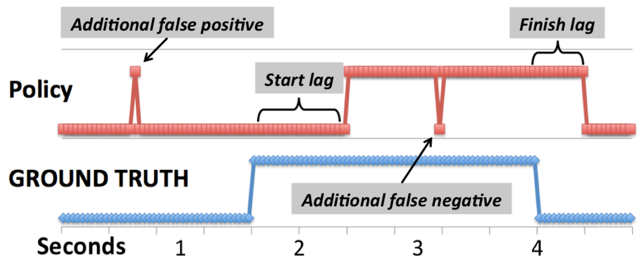


Figure 5.5: **Example Policy Evaluation.** We evaluate start lag (false negatives before a policy is correctly applied), finish lag (false positives after a policy should have been stopped), and additional false positives and negatives outside of these lags.

5.7.2 Policy Effectiveness

We evaluate policy effectiveness in representative scenarios. We aim not to maximize effectiveness but to explore policy communication technologies and demonstrate the feasibility of world-driven access control. We also view our evaluation methodology itself as a contribution for others studying access control for continuous sensing.

Evaluation Methodology

We designed and evaluated representative scenarios, each with one or more environmental policy triggers (Table 5.3). For example, in the Bathroom scenario, we affixed QR codes on and near the bathroom door, and we placed a BLE emitter and a smartphone producing a 900 Hz tone inside. In the Person scenario, a person carried a BLE emitter and wore a QR code and a red color.

We used our prototype to *record* a trace of raw recognizer events for each scenario. We constructed scenarios so that a policy was applicable in one continuous segment (e.g., a person is only seen once in the Person trace) and that the trace was realistic (e.g., we turned on a faucet in the bathroom). We then *replayed* every trace once for each policy present. Pre-recorded traces let us compare multiple policies on the same trace. Since our goal is to evaluate feasibility, we consider only one representative trace of each scenario.

We then *annotated* each trace with ground truth for each policy of interest: whether

Trace Name	*Policy Trigger*	*Intended Policy for **Targeted Event Type***	*LOC*	*Total Events*	*Length (sec.)*
Bathroom	QR Code	Block **RGB events** in bathroom.	24	10435	83
Bathroom	Bluetooth	Block **RGB events** in bathroom.	23	10435	83
Bathroom	Audio Freq. (900 Hz)	Block **RGB events** in bathroom.	35	10435	83
Person	QR Code	Remove person from **RGB events**.	84	7132	60
Person	QR Code w/ memory	Remove person from **RGB events**.	89	7132	60
Person	Bluetooth+Person	Remove person from **RGB events**.	69	7132	60
Person	Color	Remove person from **RGB events**.	104	7132	60
Speaking	Speech Keyword	Block **audio events** after keyword.	23	4529	42
Corporate	WiFi	Block **RGB events** in corporate building.	23	21064	191
Commute	Location	Blur **location events** in sensitive area.	29	475	482

Table 5.3: **Evaluated Policies.** We constructed scenarios, took traces using our prototype, and evaluated the effectiveness of various policies. In the third column, the type of event targeted by the policy is bolded; the fourth column reports total events in the trace.

the ideal enforcement of that policy would modify or block each event. We compared the results of the replay for each policy with our ground truth annotations. In particular, we introduce the following methodology for evaluating the effectiveness of a policy. Considering only those events that *may* be affected by the given policy, such as RGB in the Bathroom trace or audio in the Speech trace, we evaluate the policy as follows.

Specifically, we consider four values: (1) *start lag* (false negatives before the policy is correctly applied), (2) *finish lag* (false positives before the policy is correctly removed), (3) *additional false positives*, and (4) *additional false negatives*. Conceptually, start and finish lag measure the time it takes to recognize the policy. Additional false positives occur outside of this region and additional false negatives occur within this region; neither include the start or finish lag. Figure 5.5 shows each value visually. These values rely, of course, on specific characteristics of a trace; we attempted to construct realistic evaluation scenarios to get a conceptual idea of how well these policies work in practice.

We ran our traces with an outward-facing Kinect attached to a Lenovo W520 laptop (Core i7, 16 GB RAM, 160 GB SSD). This setup let us log traces without affecting events rates, relying on a large memory to buffer all events until trace completion. Without logging, we achieved similar rates on a less powerful Samsung tablet (Core i5, 2 GB RAM). We expect

Trace Name	*Policy Trigger*	*# Target Events*	*Start Lag*	*Finish Lag*	*Additional FN*	*Additional FP*
Bathroom	QR Code	1946	0 (0 sec)	0 (0 sec)	0	0
Bathroom	Bluetooth	1946	-50 (-2.1 sec)	183 (7.8 sec)	0	0
Bathroom	Audio Frequency (900 Hz)	1946	0 (0 sec)	0 (0 sec)	81	0
Person	QR Code	1244	279 (13.5 sec)	0 (0 sec)	88	0
Person	QR Code with memory	1244	279 (13.5 sec)	0 (0 sec)	20	0
Person	Bluetooth + Person	1244	210 (10.1 sec)	8 (0.4 sec)	0	30
Person	Color	1244	147 (6.1 sec)	0 (0 sec)	23	23
Speaking	Speech Keyword	375	16 (1.8 sec)	27 (3.0 sec)	0	0
Corporate	WiFi	2823	21 (1.4 sec)	0 (0 sec)	171	417
Commute	Location	475	14 (14.2 sec)	4 (4.1 sec)	0	0

Table 5.4: **Evaluation Results.** This table shows the results from running each policy against the corresponding trace. The first two columns match those in Table 5.3. Figure 5.5 explains how start/finish lag and additional false positives/negatives are calculated.

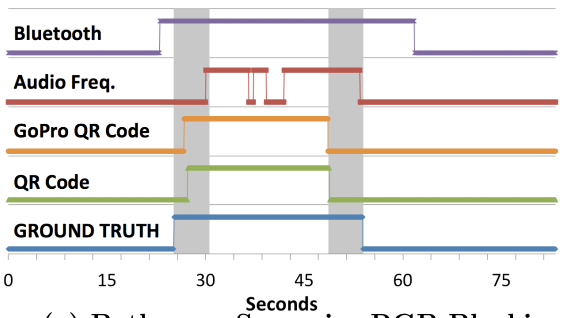
continuous sensing devices to approach the capabilities of these more powerful machines.

Evaluation Results

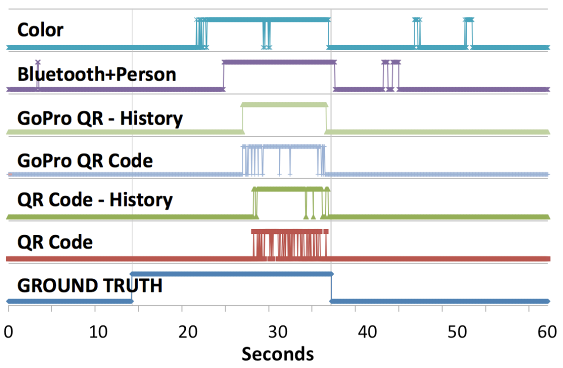
We describe lessons learned from our experience, supported by policy-specific evaluations (Table 5.4 and Figure 5.6).

Adding policy memory reduces jitter. We find that certain policy technologies show high variability in detection. For example, QR codes are detected only about every third RGB frame when present, WiFi signals are unreliable at a building's perimeter (presumably far from an access point), and audio frequency is sensitive to other ambient audio (e.g., running water in the bathroom).

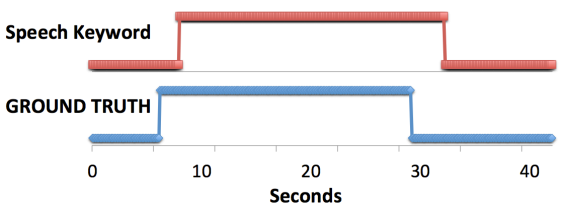
We find that we can minimize this detection jitter by adding *memory* to the policy. For example, in the original QR code policy, we consult the most recent QR code event for each RGB event to determine if the policy should be applied. In the "QR code with memory" policy, we look farther into the past, consulting up to five recent QR code events and applying the most recent policy it finds (if any). (One QR code event, or `null` if no QR code is found, is generated per RGB frame.) This change makes the QR code policy



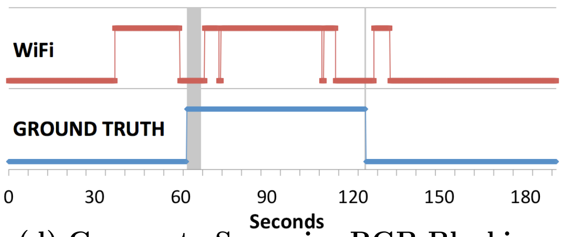
(a) Bathroom Scenario: RGB Blocking



(b) Person Scenario: RGB Modifying



(c) Speech Scenario: Audio Blocking



(d) Corporate Scenario: RGB Blocking

Figure 5.6: **Policy Evaluation.** The grey regions were annotated as "depends" in the ground truth, i.e., the human annotator believed that either blocking or not blocking the event would be acceptable; these events are not counted in Table 5.4. For reference, Figure 5.5 shows an example evaluation.

more robust: if at least one of five RGB frames results in a correctly detected QR code, the policy will not jitter in the interim. The tradeoff is that too much memory may lead to a longer finish lag or more false positives.

Hybrid techniques can improve performance. Technologies can be combined to exploit their respective strengths. For example, in the Person trace, we used Bluetooth (which has a wider range) to bootstrap person removal using color or the Kinect's person detection (which can localize the object to be modified in a frame). This result challenged our initial intuition that policies should be detectable via the same sensor as the events to which the policy applies. We reasoned that if the system is in a state where it misses a policy trigger (e.g., the camera is off), then it should also be in a state to miss the events to which the policy applies (e.g., RGB events). However, this proposal fails due to differences in the characteristics of different policy technologies.

Bootstrap remote policies early with passports. We experimented with passports in the Person trace, using Bluetooth and QR codes to load person removal policies. Passports let us easily extend our system without rebuilding it. Additionally, the latency incurred by loading policy code can be hidden by bootstrapping it as early as possible. It took on average 227.6 ms (10 trials) to load and install a passport (197.9 ms of which is network latency), increasing the start lag by 5-6 RGB frames if the policy is to be used immediately. Communicating a person's opt-out via BLE, rather than in a QR code on the person, would thus allow enough time to load the policy before the person is encountered.

Accuracy and latency may conflict. Recall that we trade off policy accuracy with performance in our implementation (Section 5.6). Rather than waiting to dispatch an RGB event until the corresponding QR event arrives, we dispatch all RGB events immediately, relying on the most recent (possibly out of date) QR code event to detect a policy. We observe that start lag consists of two components: (1) the time it takes to be in range of the policy (e.g., close enough to the QR code), and (2) the number of unmodified events after the policy comes into range but the trigger is not yet recognized. Our choice to trade off accuracy for performance potentially affects the second component.

We measured the effect of this implementation choice for QR codes. We find that the accuracy impact is negligible: in the Bathroom trace, the difference is just one frame (40 ms). That is, one RGB event contained a QR code and was let through, but the policy was applied by the next RGB event. In the Person trace, the lag is four frames (160 ms); these accuracy differences are not discernible in Figure 5.6. However, the same performance lag may noticeably impact the user in some cases: to avoid distracting lag between the real and virtual objects in augmented reality, for example, sensor input events must be dispatched in as close to real-time as possible [2]. Systems may differ in their requirements.

Better technology will improve accuracy. Because the Kinect has a relatively low RGB resolution (640x480, up to 30 frames per second), in two traces we simultaneously collected additional video using a GoPro camera (1280x1080, up to 60 fps) affixed to the Kinect (included in Figures 5.6a–b.) We synchronized Kinect and GoPro frames by hand, downsampling the GoPro points and marking the policy as enforced if any collapsed frames were blocked or modified. The barcode library could decode QR codes up to one second earlier in the GoPro traces, with fewer false negatives. The GoPro's wide-angle lens also allowed it to decode QR codes in the periphery (e.g., near but not on the bathroom door) that the Kinect did not see. Thus, as mobile cameras improve, so will their ability to detect world-driven policies.

We expect that other cases in which our system failed to match ground truth will also be improved by better technology. For example, the speech keyword recognizer is slow, resulting in large start and finish lags. Similarly, the start lag is long for all person removal policies (Figure 5.6b) because the trace involves approaching the person down a hall, and no method is effective until the camera is close enough.

World-driven access control is practical. Our experiences suggest that world-driven policies can be expressed with modest developer effort, that the system can be seamlessly extended with passports, and that policies can be detected and enforced with reasonable accuracy given the right choice of technology. Even without advances in computer vision or other technologies, world-driven access control already significantly raises the bar in some scenarios — e.g., the accuracy of our system in the bathroom setting.

5.8 Reflections and Future Work

Continuous sensing applications will become increasingly prevalent as technologies like Google Glass and Microsoft Kinect become more ubiquitous and capable. World-driven access control provides a new alternative to controlling access to sensor streams. As our work uncovers, important challenges arise when attempting to instantiate this approach in a real system. One key challenge is the tradeoff between policy detection accuracy and enforcement latency. A second key challenge is the need to verify the authenticity of signals communicated from real-world objects. Though we designed and evaluated specific approaches for overcoming both challenges, future insights may lead to alternate solutions.

We highlight these challenges in this section, with the hope of providing targets for and benefiting future researchers focused on access control for continuous sensing.

We also highlight a separate challenge: the tension between the specificity of the information broadcast in a policy and an object's privacy. While specific object descriptions (e.g., "no pictures of car with license plate 123456") are valuable both for policy verification and for extending the system's object recognition capabilities, they reveal information to anyone receiving the broadcast. We view this issue as separate from our study of how to effectively communicate and manage world-driven policies, our primary goal, but we believe it is important to consider if world-driven access control systems are deployed with highly expressive policies. Currently, policy creators must balance policy accuracy and description privacy, considering evolving social norms in the process. Since specific object descriptions in passports are nevertheless valuable for verifying that a policy has not been moved by an adversary, we encourage future work on privacy-preserving, expressive policies (e.g., encrypted policies accessible only by trusted policy modules).

A separate privacy issue arises with the use of cryptographic signatures on policies: if the signatures on each object are unique, then they can be used to track objects; we observe, however, that there are many other methods to track objects today (e.g., RFIDs, Bluetooth IDs, MAC addresses), and hence consider the issue out of scope.

Finally, as described, world-driven access control can use diverse mechanisms to communicate policies. We implemented and evaluated several approaches for explicitly communicating policies, and we encourage further study of other approaches. For example, our design is general enough to encompass *implicitly communicated* policies. Implicit policies may rely on improved computer vision (e.g., to recognize sensitive locations, as in PlaceAvoider [160]) or be inferred from natural human behavior (e.g., closing a door or whispering). Other examples of work in this area are Legion:AR [89], which uses a panel of humans to recognize activities, and work on predicting the cost of interruption from sensors [72]. Additionally, to improve policy detection accuracy, systems may include additional *policy-specific sensors*, such as cameras, whose sole purpose is to sense policies, allowing them to be lower-power than user-facing sensors [92].

5.9 Additional Related Work

We have discussed inline prior works that consider bystander privacy, allowing them to opt out with visual (e.g., [145]) or digital markers (e.g., [21, 63]). TagMeNot [23] proposes QR codes for opting out of photo tagging. These designs rely on compliant recording devices or vendors; other systems aim to prevent recording by uncooperative devices, e.g., by flashing directed light at cameras [130]. Recent work studied bystander reactions to augmented reality devices and surveyed design axes for privacy-mediation approaches [33].

Others have considered the privacy of device users. Indeed, Starner considered it a primary challenge for wearable computing [156]. Though wearable devices can improve security (e.g., with location-based or device-based access control, as in Grey [17]), data collected or sent by the device can reveal sensitive information. Such concerns motivate privacy-preserving location-based access control (e.g., [10]). Here, we assume that the device and system are trustworthy.

Others have also restricted applications' access to perceptual data. Darkly [76] integrates privacy protection into OpenCV, and the "recognizer" abstraction [75] helps limit applications' access to objects in a sensor feed; we build on these ideas. PlaceAvoider [160] identifies images captured in sensitive locations and withholds them from applications. PiBox [90] restricts applications from secretly leaking private data, but provides weak guarantees when users explicitly share it. Our approach helps users avoid accidentally sharing sensitive content with untrusted applications.

5.10 Conclusion

This chapter has revisited the key challenge of application permission granting identified in Chapter 1, and first considered in Chapter 3, in a new technological context. Continuous sensing applications on platforms like smartphones, Google Glass, and XBox Kinect raise serious access control challenges not solved by current techniques, including user-driven access control (Chapter 3). This chapter introduced *world-driven access control*, by which real-world objects specify per-application privacy policies. This approach aims to enable permission management at the granularity of objects rather than complete sensor streams, and without explicitly involving the user. A trusted platform detects these policies in the environment and automatically adjusts each application's "view" accordingly.

We built an end-to-end world-driven access control prototype, surfacing key challenges. We introduced *passports* to address policy authenticity and to allow our system to be dynamically extended. We mitigated detection inaccuracies by combining multiple policy communication techniques and introducing memory into policies. Finally, we explored the tradeoffs between policy accuracy and performance, and between broadcast policies and privacy.

Our prototype enforces many policies, each expressed with modest developer effort, with reasonable accuracy even with today's technologies and off-the-shelf algorithms. Our experiences suggest that world-driven access control can relieve the user's permission management burden while preserving functionality and protecting privacy. Beyond proposing and evaluating a new design point for access control in continuous sensing, this work represents a step toward *integrating physical objects into a virtual world*; we believe our efforts lay the groundwork for future work in this space.

Chapter 6
CONCLUSION

Modern and emerging client platforms, including browsers, modern operating systems like smartphones, and emerging platforms like augmented reality systems, provide great benefits to their users. Unfortunately, the untrusted applications that users may install on these platforms come with security and privacy risks. For example, advertisements on websites may track users' browsing behaviors, smartphone malware may secretly send costly SMS messages, and even legitimate applications on smartphones and other platforms may leak or misuse users' private data (e.g., sending the user's location to third-party servers).

This dissertation identified and characterized two fundamental security and privacy challenges due to these untrusted applications: (1) embedded third-party principals, which come with security and privacy implications for both the embedded content and the host applications, and (2) application permission granting, where users are asked to make decisions about which permissions an application should receive, for which today's approaches are neither sufficiently usable nor secure. This dissertation explored and addressed these challenges in the contexts of browsers and the web, modern operating systems such as smartphones, and emerging continuous sensing and augmented reality platforms.

With respect to the first challenge — embedded third-party principals — this dissertation first studied third-party tracking enabled by embedded content on the web (Chapter 2), and then considered more generally a set of requirements and techniques for securing embedded user interfaces (Chapter 4). With respect to the second challenge — application permissions — this dissertation leveraged secure embedded user interfaces to enable user-driven access control as a permission granting model for today's smartphone applications (Chapter 3), and then developed world-driven access control as a new permission granting model for emerging continuous sensing applications in which explicit user actions are infrequent (Chapter 5). The ideas presented in this dissertation have resulted in several concrete software artifacts that have been made available to the community: TrackingObserver, a browser-based web tracking detection platform; ShareMeNot, a new defense for social media widget trackers that protects privacy without compromising the functionality of the widgets; and LayerCake, a modified version of Android that securely supports cross-principal user interface embedding, enabling, among other things, the access control gadgets central to user-driven access control. Together, these works mitigate many of the risks posed by untrusted applications and provide a foundation for addressing two fundamental challenges in computer security and privacy for modern and emerging client platforms.

BIBLIOGRAPHY

[1] ZXing.Net. `http://zxingnet.codeplex.com/`.

[2] Michael Abrash. Latency – the sine qua non of AR and VR, 2012. `http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/`.

[3] Güneş Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the web for fingerprinters. In *20th ACM Conference on Computer and Communications Security*. ACM, 2013.

[4] Ada Initiative. Another way to attract women to conferences: photography policies, 2013. `http://adainitiative.org/2013/07/another-way-to-attract-women-to-conferences-photography-policies/`.

[5] Adobe. User-initiated action requirements in Flash Player 10. `http://www.adobe.com/devnet/flashplayer/articles/fplayer10_uia_requirements.html`, 2008.

[6] Gaurav Aggrawal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the USENIX Security Symposium*, 2010.

[7] Istemi Ekin Akkus, Ruichuan Chen, Michaela Hardt, Paul Francis, and Johannes Gehrke. Non-tracking web analytics. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[8] Apple. App Sandbox and the Mac App Store. `https://developer.apple.com/videos/wwdc/2011/` Nov. 2011.

[9] Apple. iOS. `http://www.apple.com/ios/`.

[10] Claudio Agostino Ardagna, Marco Cremonini, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Privacy-enhanced Location-based Access Control. In *Handbook of Database Security*, pages 531—552. 2008.

[11] R. B. Arthur and D. R. Olsen. Privacy-aware shared UI toolkit for nomadic environments. In *Software Practice and Experience*, 2011.

[12] Mika Ayenson, Dietrich James Wambach, Ashkan Soltani, Nathan Good, and Chris Jay Hoffnagle. Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning. *Social Science Research Network Working Paper Series*, 2011.

[13] Mario Ballano. Android Threats Getting Steamy. Symantec Official Blog, Febuary 2011. `http://www.symantec.com/connect/blogs/android-threats-getting-steamy`.

[14] Adam Barth. Security in Depth: HTML5s @sandbox, 2010. `http://blog.chromium.org/2010/05/security-in-depth-html5s-sandbox.html`.

[15] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, February 2010.

[16] Jason Bau, Jonathan Mayer, Hristo Paskov, and John C. Mitchell. A Promising Direction for Web Tracking Countermeasures. In *Web 2.0 Security and Privacy*, 2013.

[17] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the Information Security Conference*, pages 432–445. Springer Verlag LNCS, 2005.

[18] Ole Begemann. Remote View Controllers in iOS 6, October 2012. `http://oleb.net/blog/2012/02/what-ios-should-learn-from-android-and-windows-8/`.

[19] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *Proceedings of the ACM Conference on Computer Graphics (SIGGRAPH)*, 1993.

[20] Nikita Borisov and Eric A. Brewer. Active certificates: A framework for delegation. In *Network and Distributed System Security Symposium (NDSS)*, 2002.

[21] Jack Brassil. Technical Challenges in Location-Aware Video Surveillance Privacy. In Andrew Senior, editor, *Protecting Privacy in Video Surveillance*, pages 91–113. Springer-Verlag, 2009.

[22] butterscotch.com. CES: Wearable HD 3D cameras by GOPRO, 2011. `http://youtu.be/w1p7yvb57Tg`.

[23] Alberto Cammozzo. TagMeNot, 2011. `http://tagmenot.info/`.

[24] Eun Kyoung Choe, Sunny Consolvo, Jaeyeon Jung, Beverly L. Harrison, and Julie A. Kientz. Living in a glass house: a survey of private moments in the home. In *ACM UbiComp*, 2011.

[25] Eun Kyoung Choe, Sunny Consolvo, Jaeyeon Jung, Beverly L. Harrison, Shwetak N. Patel, and Julie A. Kientz. Investigating receptiveness to sensing and inference in the home using sensor proxies. In *ACM UbiComp*, 2012.

[26] Chromium. Security Issues. `https://code.google.com/p/chromium/issues/list?`

q=label:Security, February 2011.

[27] Jeremy Clark and Paul C. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. *IEEE Symposium on Security & Privacy*, 2013.

[28] CNET.com. Hands on with google glass, 2013. http://www.youtube.com/watch?v=n7XGk5BYNIo.

[29] CNXSoft. Qualcomm fast computer vision sdk, 2011. http://www.cnx-software.com/2011/10/28/qualcomm-fast-computer-vision-sdk/.

[30] Sunny Consolvo, Jaeyeon Jung, Ben Greenstein, Pauline Powledge, Gabriel Maganis, and Daniel Avrahami. The Wi-Fi Privacy Ticker: Improving Awareness and Control of Personal Information Exposure on Wi-Fi. In *12th ACM Confference on Ubiquitous Computing*, 2010.

[31] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A Safety-Oriented Platform for Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[32] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. Baaz: A System for Detecting Access Control Misconfigurations. In *Proceedings of the USENIX Security Conference*, 2010.

[33] Tamara Denning, Zakariya Dehlawi, and Tadayoshi Kohno. In situ with bystanders of augmented reality glasses: Perspectives on recording and privacy-mediating technologies. In *ACM CHI*, 2014.

[34] Mohan Dhawan, Christian Kreibich, and Nicholas Weaver. The Priv3 Firefox Extension. http://priv3.icsi.berkeley.edu/.

[35] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium*, 2011.

[36] M. Dixon and J. Fogarty. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2010.

[37] Dowdell, John. Clipboard pollution. http://blogs.adobe.com/jd/2008/08/clipboard_pollution.html, 2008.

[38] J. R. Eagan, M. Beaudouin-Lafon, and W. E. Mackay. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the ACM Symposium on User*

Interface Software and Technology, 2011.

[39] Peter Eckersley. How unique is your web browser? In *Proceedings of the International Conference on Privacy Enhancing Technologies*, 2010.

[40] W. K. Edwards, S. Hudson, J. Marinacci, R. Rodenstein, T. Rodriguez, and I. Smith. Systematic Output Modification in a 2D User Interface Toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 1997.

[41] S. Egelman, L. F. Cranor, and J. Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2008.

[42] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Conference on Operating System Design and Implementation*, 2010.

[43] Jeremy Epstein, John McHugh, and Rita Pascale. Evolution of a Trusted B3 Window System Prototype. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1992.

[44] Mathias Ettrich and Owen Taylor. XEmbed Protocol Specification, 2002. `http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html`.

[45] Christian Eubank, Marcela Melara, Diego Perez-Botero, and Arvind Narayanan. Shining the Floodlights on Mobile Web Tracking — A Privacy Survey. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy*, 2013.

[46] Facebook. Social Plugins. `https://developers.facebook.com/docs/plugins/`.

[47] Facebook. Like button requires confirm step, 2012. `https://developers.facebook.com/bugs/412902132095994/`.

[48] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How To Ask For Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security*, 2012.

[49] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The Effectiveness of Application Permissions. In *Proceedings of the USENIX Conference on Web Application Development*, June 2011.

[50] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attentions, Comprehension, and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security*, 2012.

[51] Norman Feske and Christian Helmuth. A Nitpicker's Guide to a Minimal-Complexity Secure GUI. In *Proceedings of the Annual Computer Security Applications Conference*, 2005.

[52] Matthew Fredrikson and Benjamin Livshits. RePriv: Re-Envisioning In-Browser Privacy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[53] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[54] James Gettys and Keith Packard. The X Window System. *ACM Transactions on Graphics*, 5:79–109, 1986.

[55] Avi Goldfarb and Catherine E. Tucker. Privacy Regulation and Online Advertising. *Management Science*, 57(1), January 2011.

[56] Nathaniel Good, Rachna Dhamija, Jens Grossklags, David Thaw, Steven Aronowitz, Deirdre Mulligan, and Joseph Konstan. Stopping spyware at the gate: a user study of privacy, notice and spyware. In *Proceedings of the Symposium on Usable Privacy and Security*, 2005.

[57] Google. AdMob Ads SDK. `https://developers.google.com/mobile-ads-sdk/`.

[58] Google. Android OS. `http://www.android.com/`.

[59] Google. Google Glass. `http://www.google.com/glass/`.

[60] Richard Gray. The places where Google Glass is banned, December 2013. `http://www.telegraph.co.uk/technology/google/10494231/The-places-where-Google-Glass-is-banned.html`.

[61] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web Browsing with the OP Web Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[62] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[63] J. Alex Halderman, Brent Waters, and Edward W. Felten. Privacy Management for Portable Recording Devices. In *Proceedings of the ACM Workshop on Privacy in Electronic Society*, 2004.

[64] Seungyeop Han, Jaeyeon Jung, and David Wetherall. A Study of Third-Party Tracking by Mobile Apps in the Wild. Technical Report UW-CSE-12-03-01, University of Washington, March 2012.

[65] Seungyeop Han, Vincent Liu, Qifan Pu, Simon Peter, Thomas E. Anderson, Arvind Krishnamurthy, and David Wetherall. Expressive privacy control with pseudonyms. In *SIGCOMM*, 2013.

[66] B. Hartmann, L. Wu, K. Collins, and S. R. Klemmer. Programming by a Sample: Rapidly Creating Web Applications with d.mix. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2007.

[67] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[68] J. Howell and S. Schechter. What You See Is What They Get: Protecting Users from Unwanted Use of Microphones, Camera, and Other Sensors. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop*, 2010.

[69] Jon Howell, Bryan Parno, and John R. Douceur. Embassies: Radically Refactoring the Web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[70] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, 2012.

[71] S. Hudson, J. Mankoff, and I. Smith. Extensible Input Handling in the subArctic Toolkit. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2005.

[72] Scott Hudson, James Fogarty, Christopher Atkeson, Daniel Avrahami, Jodi Forlizzi, Sara Kiesler, Johnny Lee, and Jie Yang. Predicting human interruptibility with sensors: a wizard of oz feasibility study. In *ACM CHI*, 2003.

[73] Sunghwan Ihm and Vivek Pai. Towards Understanding Modern Web Trafc. In *Proceedings of the ACM Internet Measurement Conference*, 2011.

[74] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting Browser State From Web Privacy Attacks. In *Proceedings of the International World Wide Web Conference*, 2006.

[75] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J. Wang, and Eyal Ofek. Enabling Fine-Grained Permissions for Augmented Reality Applications with Recognizers. In *Proceedings of the USENIX Security Symposium*, 2013.

[76] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[77] A. Janc and L. Olejnik. Feasibility and Real-World Implications of Web Browser History Detection. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy*, 2010.

[78] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[79] Carlos Jensen, Chandan Sarkar, Christian Jensen, and Colin Potts. Tracking website data-collection and privacy practices with the iWatch web crawler. In *Proceedings of the Symposium on Usable Privacy and Security*, 2007.

[80] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[81] Doug Kilpatrick, Wayne Salamon, and Chris Vance. Securing the X Window System with SELinux. Technical Report 03-006, NAI Labs, March 2003.

[82] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2005.

[83] Georgios Kontaxis, Michalis Polychronakis, Angelos D. Keromytis, and Evangelos P. Markatos. Privacy-preserving social plugins. In *USENIX Security Symposium*, 2012.

[84] Munir Kotadia. Jamming device aims at camera phones, 2003. `http://news.cnet.com/Jamming-device-aims-at-camera-phones/2100-1009_3-5074852.html`.

[85] Balachander Krishnamurthy, Konstantin Naryshkin, and Craig Wills. Privacy Leakage vs. Protection Measures: The Growing Disconnect. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy*, 2011.

[86] Balachander Krishnamurthy and Craig Wills. On the leakage of personally identifiable information via online social networks. In *Proceedings of the ACM Workshop on Online Social Networks*, 2009.

[87] Balachander Krishnamurthy and Craig Wills. Privacy Diffusion on the Web: a Longitudinal Perspective. In *Proceedings of the International World Wide Web Conference*, 2009.

[88] Balachander Krishnamurthy and Craig E. Wills. Generating a Privacy Footprint on

the Internet. In *Proceedings of the ACM Internet Measurement Conference*, 2006.

[89] Walter Lasecki, Young Chol Song, Henry Kautz, and Jeffrey Bigham. Real-time crowd labeling for deployable activity recognition. In *Computer Supported Cooperative Work (CSCW)*, 2013.

[90] S. Lee, E. Wong, D. Goel, M. Dahlin, and V. Shmatikov. PiBox: A platform for privacy preserving apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[91] Pedro G. Leon, Blase Ur, Yang Wang, Manya Sleeper, Rebecca Balebako, Richard Shay, Lujo Bauer, Mihai Christodorescu, and Lorrie Faith Cranor. What Matters to Users? Factors that Affect Users' Willingness to Share Information with Online Advertisers. In *Symposium on Usable Privacy and Security*, 2013.

[92] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. Energy characterization & optimization of image sensing toward continuous mobile vision. In *MobiSys*, 2013.

[93] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-User Programming of Mashups with Vegemite. In *Proceedings of the ACM Conference on Intelligent User Interfaces*, 2009.

[94] Antonio Lioy and Gianluca Ramunno. Trusted computing. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, pages 697–717. 2010.

[95] H. R. Lipford, J. Watson, M. Whitney, K. Froiland, and R. W. Reeder. Visual vs. Compact: A Comparison of Privacy Policy Interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2010.

[96] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. BLADE: An Attack-Agnostic Approach For Preventing Drive-By Malware Infections. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[97] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android System. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[98] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of the International Symposium on Foundations and Practice of Security*, 2012.

[99] I. S. MacKenzie. Fitts' Law as a Research and Design Tool in Human-Computer Interaction. *Human-Computer Interaction (HCI)*, 7(1):91–139, 1992.

[100] Michelle Madejski, Maritza Johnson, and Steven M. Bellovin. A Study of Privacy Settings Errors in an Online Social Network. In *Proceedings of the IEEE International Workshop on Security and Social Networking*, 2012.

[101] Moxie Marlinspike. Convergence. `http://convergence.io/`.

[102] Michael Massimi, Khai N. Truong, David Dearman, and Gillian R. Hayes. Understanding recording technologies in everyday life. *IEEE Pervasive Computing*, 9(3):64–71, 2010.

[103] Jonathan Mayer. Tracking the Trackers: Early Results, July 2011. `http://cyberlaw.stanford.edu/node/6694`.

[104] Jonathan Mayer. Tracking the Trackers: Self Help Tools, September 2011. `http://cyberlaw.stanford.edu/blog/2011/09/tracking-trackers-self-help-tools`.

[105] Jonathan Mayer. Safari tracking, January 2012. `http://webpolicy.org/2012/02/17/safari-trackers/`.

[106] Jonathan Mayer, Arvind Narayanan, and Sid Stamm. Do Not Track: A Universal Third-Party Web Tracking Opt Out. Internet-Draft, March 2011. `http://tools.ietf.org/html/draft-mayer-do-not-track-00`.

[107] Jonathan R. Mayer and John C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[108] Aleecia M. McDonald and Lorrie Faith Cranor. Americans' Attitudes about Internet Behavioral Advertising Practices. In *Proceedings of the Workshop on Privacy in the Electronic Society*, 2010.

[109] Meta. Spaceglasses, 2013. `http://spaceglasses.com`.

[110] Microsoft. Accessing files with file pickers. `http://msdn.microsoft.com/en-us/library/windows/apps/hh465174.aspx`.

[111] Microsoft. Application Domains. `http://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.110).aspx`.

[112] Microsoft. User Account Control. `microsoft.com/en-us/library/windows/desktop/aa511445.aspx`.

[113] Microsoft. How to Prevent Web Sites From Obtaining Access to the Contents of Your Windows Clipboard, March 2007. `http://support.microsoft.com/kb/224993`.

[114] Microsoft. Silverlight Clipboard Class, 2010. `http://msdn.microsoft.com/en-us/library/system.windows.clipboard%28v=VS.95%29.aspx`.

[115] Microsoft. Creating your own code access permissions, 2013. http://msdn.microsoft.com/en-us/library/yctbsyf4(v=vs.90).aspx.

[116] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[117] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. SpyProxy: Execution-Based Detection of Malicious Web Content. In *Proceedings of the USENIX Security Symposium*, 2007.

[118] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In *Proceedings of the Symposium on Usable Privacy and Security*, 2010.

[119] Mozilla. Bug 417800 — Revert to not blocking third-party cookies, 2008. https://bugzilla.mozilla.org/show_bug.cgi?id=417800.

[120] Mozilla Foundation. Known Vulnerabilities in Mozilla Products, February 2011. http://www.mozilla.org/security/known-vulnerabilities/.

[121] B. Myers, S. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. In *ACM Transactions on Computer-Human Interaction*, 2000.

[122] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[123] Brook Novak. Accessing the System Clipboard with JavaScript: A Holy Grail? http://brooknovak.wordpress.com/2009/07/28/accessing-the-system-clipboard-with-javascript/.

[124] NSA Central Security Service. Security-Enhanced Linux. http://www.nsa.gov/research/selinux/, January 2009.

[125] Kevin O'Brien. Swiss Court Orders Modifications to Google Street View, 2012. http://www.nytimes.com/2012/06/09/technology/09iht-google09.html.

[126] D. R. Olsen, Jr. Evaluating User Interface Systems Research. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2007.

[127] Matthew Panzarino. Inside the revolutionary 3d vision chip at the heart of googles project tango phone, February 2014. http://tcrn.ch/1fkCuWK.

[128] Jithendra K. Paruchuri, Sen-Ching S. Cheung, and Michael W. Hail. Video data hiding for managing privacy information in surveillance systems. *EURASIP Journal*

on Info. Security, pages 7:1–7:18, January 2009.

[129] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A Picture of Search. In *Proceedings of the Conference on Scalable Information Systems*, 2006.

[130] Shwetak N. Patel, Jay W. Summet, and Khai N. Truong. BlindSpot: Creating Capture-Resistant Spaces. In Andrew Senior, editor, *Protecting Privacy in Video Surveillance*, pages 185–201. Springer-Verlag, 2009.

[131] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012.

[132] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[133] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan, and Seth J. Teller. The cricket compass for context-aware mobile applications. In *Mobile Computing and Networking*, 2001.

[134] Quest Visual. Word Lens Application. `http://questvisual.com/`.

[135] Charles Reis and Steven D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2009.

[136] Franziska Roesner, James Fogarty, and Tadayoshi Kohno. User Interface Toolkit Mechanisms for Securing Interface Elements. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2012.

[137] Franziska Roesner and Tadayoshi Kohno. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the USENIX Security Symposium*, 2013.

[138] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[139] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[140] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and He-

len J. Wang. World-Driven Access Control for Continuous Sensing Applications. Technical Report MSR-TR-2014-67, Microsoft Research, May 2014.

[141] Franziska Roesner, Christopher Rovillos, Tadayoshi Kohno, and David Wetherall. ShareMeNot: Balancing Privacy and Functionality of Third-Party Social Widgets. *USENIX ;login:*, 37, 2012.

[142] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS. Invisible Things Lab. `http://qubes-os.org`.

[143] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: A Study of Clickjacking Vulnerabilities on Popular Sites. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy*, 2010.

[144] S.E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[145] Jeremy Schiff, Marci Meingast, Deirdre K. Mulligan, Shankar Sastry, and Kenneth Y. Goldberg. Respectful Cameras: Detecting Visual Markers in Real-Time to Address Privacy Concerns. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 2007.

[146] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[147] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS Trusted Window System. In *Proceedings of the USENIX Security Symposium*, 2004.

[148] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[149] Jeff Shirley and David Evans. The User is Not the Enemy: Fighting Malware by Tracking User Intentions. In *Proceedings of the New Security Paradigms Workshop*, 2008.

[150] 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 & Pattern Recognition*, 2011.

[151] Tom Simonite. Popular Ad Blocker Also Helps the Ad Industry, June 2013. `http://mashable.com/2013/06/17/ad-blocker-helps-ad-industry/`.

[152] Natasha Singer. Do-Not-Track Talks Could Be Running Off the Rails. The New York Times, May 2013. `http://bits.blogs.nytimes.com/2013/05/03/do-not-track-talks-could-be-running-off-the-rails/`.

[153] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. Flash Cookies and Privacy. *Social Science Research Network Working Paper Series*, August 2009.

[154] Sophos. Facebook Worm: Likejacking, 2010. `http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/`.

[155] Sophos. Security threat report, 2014. `http://www.sophos.com/en-us/threat-center/security-threat-report.aspx`.

[156] T.E. Starner. The Challenges of Wearable Computing: Part 2. *IEEE Micro*, 21(4):54—67, 2001.

[157] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. Polaris: Virus-Safe Computing for Windows XP. *Communications of the ACM*, 49:83–88, September 2006.

[158] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the USENIX Security Symposium*, 2009.

[159] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[160] Robert Templeman, Mohammed Korayem, David Crandall, and Apu Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[161] Robert Templeman, Zahid Rahman, David J. Crandall, and Apu Kapadia. PlaceRaider: Virtual Theft in Physical Spaces with Smartphones. *CoRR*, abs/1209.5982, 2012.

[162] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35:80–86, 1997.

[163] The 5 Point Cafe. Google Glasses Banned, March 2013. `http://the5pointcafe.com/google-glasses-banned/`.

[164] ThreatMetrix. Tech. overview. `http://threatmetrix.com/technology/`

technology-overview/.

[165] Tom Simonite. Bringing cell-phone location-sensing indoors. http://bit.ly/TVyMEx.

[166] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[167] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *8th Symposium on Usable Privacy and Security*, 2012.

[168] Robert Vamosi. Device Fingerprinting Aims To Stop Online Fraud. PCWorld, March 2009. http://www.pcworld.com/businesscenter/article/161036/.

[169] W3C. Same Origin Policy. http://www.w3.org/Security/wiki/Same_Origin_Policy.

[170] W3C. Device API Working Group, 2011. http://www.w3.org/2009/dap/.

[171] W3C. Device APIs and Policy Working Group, 2011. http://www.w3.org/2009/dap/.

[172] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the USENIX Security Symposium*, 2009.

[173] Helen J. Wang, Alexander Moshchuk, and Alan Bush. Convergence of Desktop and Web Applications on a Multi-Service OS. In *Proceedings of the USENIX Workshop on Hot Topics in Security*, 2009.

[174] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Security Symposium*, 2008.

[175] J. Wong and J. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2007.

[176] J. P. L. Woodward. Security Requirements for System High and Compartmented Mode Workstations. Technical Report MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as DDS-2600-5502-87), The MITRE Corporation, November 1987.

[177] Ka-Ping Yee. User Interaction Design for Secure Systems. In *Proceedings of the ACM Conference on Information and Communications Security*, 2002.

[178] Ka-Ping Yee. Aligning Security and Usability. *IEEE Security and Privacy*, 2(5):48–55, September 2004.

[179] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[180] Harlan Yu. Do Not Track: Not as Simple as it Sounds, August 2010. `https://freedom-to-tinker.com/blog/harlanyu/do-not-track-not-simple-it-sounds`.

[181] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[182] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995.