This paper appears at the 33rd IEEE Symposium on Security and Privacy (Oakland 2012).

User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems

Franziska Roesner, Tadayoshi Kohno
{*franzi, yoshi*}*@cs.washington.edu*
University of Washington

Alexander Moshchuk, Bryan Parno, Helen J. Wang
{*alexmos, parno, helenw*}*@microsoft.com*
Microsoft Research

Crispin Cowan
crispin@microsoft.com
Microsoft

***Abstract*— Modern client platforms, such as iOS, Android, Windows Phone, Windows 8, and web browsers, run each application in an isolated environment with limited privileges. A pressing open problem in such systems is how to allow users to grant applications access to *user-owned resources*, e.g., to privacy- and cost-sensitive devices like the camera or to user data residing in other applications. A key challenge is to enable such access in a way that is non-disruptive to users while still maintaining least-privilege restrictions on applications.**

In this paper, we take the approach of *user-driven access control*, whereby permission granting is built *into* existing user actions in the context of an application, rather than added as an afterthought via manifests or system prompts. To allow the system to precisely capture permission-granting intent in an application's context, we introduce *access control gadgets* (ACGs). Each user-owned resource exposes ACGs for applications to embed. The user's authentic UI interactions with an ACG grant the application permission to access the corresponding resource. Our prototyping and evaluation experience indicates that user-driven access control enables in-context, non-disruptive, and least-privilege permission granting on modern client platforms.

1 Introduction

Many modern client platforms treat applications as distinct, untrusted principals. For example, smartphone operating systems like Android [2] and iOS [4] isolate applications into separate processes with different user IDs, and web browsers implement the same-origin policy [28], 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 [2, 5], or they unintelligently prompt users to determine their intent [4, 21].

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 [22, 32, 33] applied this principle largely in the context of least-privilege file picking, where an application can access only user-selected files; in this paper, 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 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 [37] 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.

2 State of the Art in Permission Granting

We motivate our work by identifying shortcomings in existing permission granting systems.

Global Resources. Traditional desktop systems expose user-owned resources to applications simply by globalizing them. Similarly, smartphone OSes expose a global clipboard to applications. While user-friendly in the benign case, this model violates least-privilege and allows unintended accesses (e.g., [9]). Our user studies (Section 6.2) indicate that such exposures contradict users' expectations (e.g., users expect data on the clipboard to remain private until pasted).

Manifests. Applications in Android [2] and Facebook [10] use install-time manifests to request access to user-owned resources. Once the user agrees, the installed application has *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. Furthermore, studies indicate that many applications ask for more permissions than needed [6, 11] and that users pay little attention to, and show little comprehension of, Android manifests [12]. Recent Android malware outbreaks suggest that users install applications that ask for excessive permissions [5].

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.

Prompts. By contrast, iOS [4] prompts users the first time an application wishes to access a resource. Windows displays a User Account Control prompt [21] when an application requires additional privileges to alter the system. Nascent support for user-owned resources in HTML5 involves prompting for geolocation access.

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 [23, 39].

No Access. Some systems simply do not support application access to user-owned resources. For example, today's web applications generally cannot access a user's local devices. Browser plugins are an exception, but they have access to all user-owned resources, which violates least-privilege. Existing specifications for permitting access to user-owned resources [35] 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. Research browsers and browser operating systems [8, 15, 34, 36, 37] also have not addressed access control for user-owned resources.

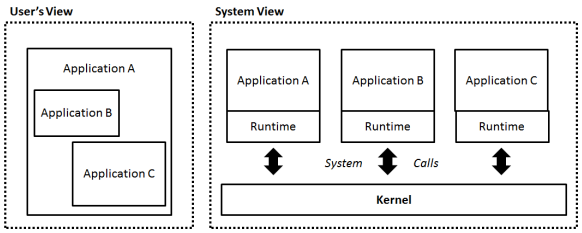

Figure 1: **Application Model.** *As in web mashups, applications may embed other applications. Each application is isolated and runs atop a generic runtime (e.g., a browser renderer or Win32 library).*

3 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).

3.1 System Model

We assume (see Figure 1) 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) [31, 36]. Furthermore, the kernel dispatches UI events only to the application with which the user is interacting. The prototype system [37] that we use to implement and evaluate user-driven access control supports these properties.

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.

[1] Research browser Gazelle [36] 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.

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 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., [27, 30]). 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. **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 [40] 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.

[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).

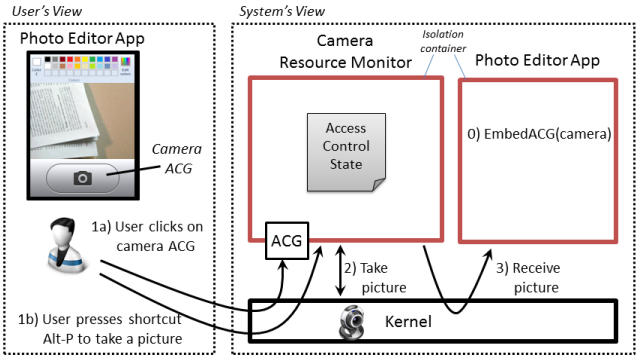

Figure 2: **System Overview.** *Resource monitors mediate access to user-owned resources. They capture the user's intent to grant an application access to that resource via (1a) UI elements in the form of access control gadgets, and/or via (1b) permission-granting input sequences detected by the kernel.*

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

Figure 2 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 2) 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.

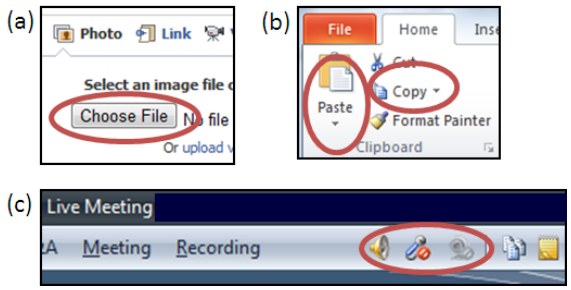

Figure 3: **Example Access Control Gadgets (ACGs).** *The circled UI elements are application-specific today, but in our model, they would be ACGs for (a) content picking, (b) copy & paste, or (c) device access.*

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 directly to grant the permission to take a picture (arrow 1b in Figure 2).

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 gives a few examples. We later describe our implementation of a representative set of ACGs in Section 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 4.1); we defer the design of specific ACGs to Section 5.3. Our system supports a variety of access semantics (Section 4.2), such as one-time, session, and permanent access grants; it also supports ACG composition (Section 4.3) to reduce the number of user actions required for granting permissions. In Section 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 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.

4.1 Capturing User Intent with ACGs

To accurately capture the user's permission-granting intent, the kernel must provide a trusted path [38] 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. 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.

4.1.1 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 3.1), 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 [19]). 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 [7] (Issue 52868) and Firefox [24] (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 6.7), we find that gadgets with limited customization would be sufficient in most cases.

4.1.2 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.

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
```

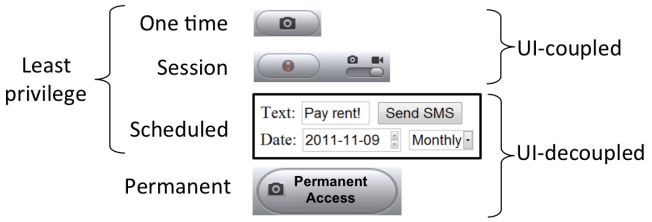

Figure 4: **Access Semantics.** *Access duration, relationship with user actions, and security implications.*

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-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.

4.2 Access Semantics

After capturing a user's intent to grant access — via an ACG or a permission-granting input sequence (Section 4.5) — a resource monitor must act on this intent. We discuss the mechanisms for granting access in our implementation in Section 5; here we consider the semantics of different types of access.

Figure 4 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*.

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 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 [17] 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 [16].

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 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.

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 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 4.1.2).

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 7) has suggested specific reserved gestures [14, 29, 31], 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 4.1.2, 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 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.

5 Implementation

We build our new mechanisms into ServiceOS [37], a prototype system that provides the properties described in Section 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 5.1), and **(2)** the appropriate RM to act on that intent (Section 5.2). Tables 1 and 2 summarize the system calls and application upcalls we implemented to support user-driven access control. Only the kernel may issue upcalls to applications. Section 5.3 describes end-to-end scenarios via representative ACGs.

5.1 Capturing User Intent

5.1.1 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.

5.1.2 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.

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 an 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 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.*

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 the object under the cursor is draggable
upcall	`CheckNestedPermission(windowId, nestedApp, acgType)`	Determines if a nested application may embed an ACG

Table 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.*

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.

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).

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/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).

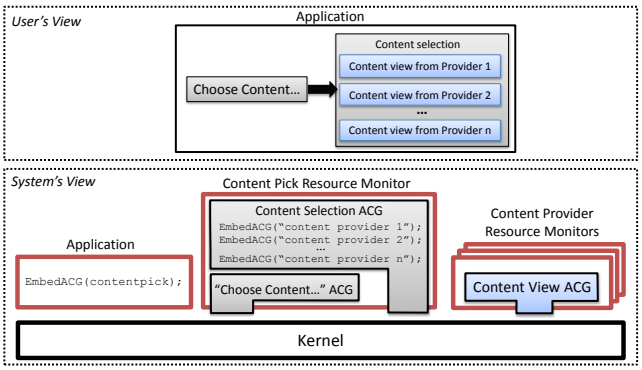

Figure 5: **Content Picking via ACGs.** *An application may embed a content-picking ACG (the "Choose content..." button above). When the user clicks on this button, the content-picking RM presents a content-selection ACG, which displays available content by embedding application-specific ACGs from each content provider. The user may select content from one of these content-view ACGs, thereby enabling the original embedding application to access it.*

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 5 illustrates our design for content picking. Each content provider exposes an application-specific ACG (Section 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 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.

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 6.1). **(2)** Unlike existing models, our least-privilege model matches users' expectations (Section 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 6.3). Indeed, **(4)** attackers have only very limited ways of gaining unauthorized access in our system (Section 6.4). We find that **(5)** system developers can easily add new resources (Section 6.5), that **(6)** application developers can easily incorporate user-driven access control (Section 6.6), and that **(7)** our design does not unreasonably restrict customization (Section 6.7). **(8)** Finally, the performance impact is negligible (Section 6.8).

Class of Vulnerability	Example	% We Eliminate by Design	
		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: **Relevant browser vulnerabilities.** *We categorize Chrome [7] and Firefox [24] vulnerabilities that affect user-owned resources; we show the percentage of vulnerabilities that user-driven access control eliminates by design.*

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) [7] and Firefox (v. 2.0-3.6) [24]. 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., [15, 36]) 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 sub-categorize these remaining vulnerabilities and analyze which can be eliminated by user-driven access control. Table 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 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), 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.

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

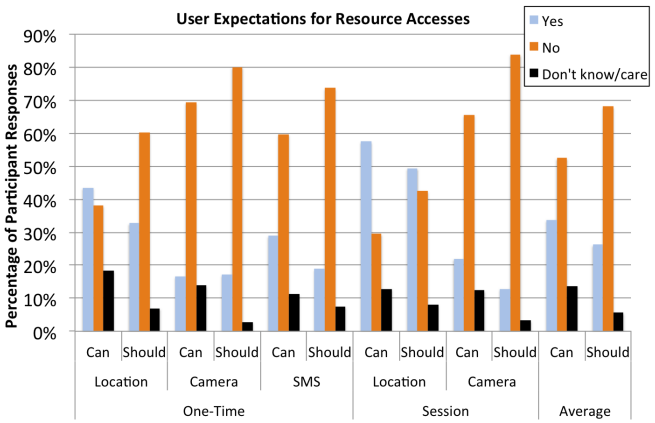

Figure 6: **User Expectations of One-Time and Session Access.** *For one-time (location, camera, SMS) and session-based (location, camera) access scenarios, participants were asked (1)* ***can*** *the application access the resource before they press the button, and (2)* ***should*** *it be able to access the resource before they press the button.*

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 6 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.

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 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.

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 4-A summarizes these results. By using an ACG to guide these 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:

	Access Semantics (A)					Customization (B)	
	UI-Decoupled		*UI-Coupled*	*Unrelated to primary*			
	(Permanent)	*(Scheduled)*	*(One-Time/Session)*	*functionality (e.g., ads)*	*Unknown*	*Limited*	*Arbitrary*
Camera	0	0	7	0	2	4	1
Location (GPS)	1	3	13	3	1	5	5
Location (Network)	0	3	9	8	2		
Microphone	0	0	14	0	0	6	4
Send SMS	0	0	3	0	2	3	0
Receive SMS	5	0	0	0	0	N/A	N/A
Read SMS	2	2	3	0	2	0	2
Make Calls	0	0	8	0	2	8	0
Intercept Outgoing Calls	1	0	0	0	0	N/A	N/A
Read Contacts	2	2	18	0	2	14	2
Write Contacts	0	0	13	0	2		
Read Calendar	1	1	0	0	1	0	0
Write Calendar	0	0	1	0	3	0	1
Total	12 (8%)	11 (8%)	89 (62%)	11 (8%)	20 (14%)	40 (73%)	15 (27%)

Table 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 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 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.*

- *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 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.

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 [23]. 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 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 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 5: **Evaluation of Extensibility and Ease of Use.** *This table shows the approximate lines of C# code added to the system (Table 5-A) and to applications (Table 5-B).*

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 5-A shows the development effort required to implement the example ACGs described in Section 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.

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 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.

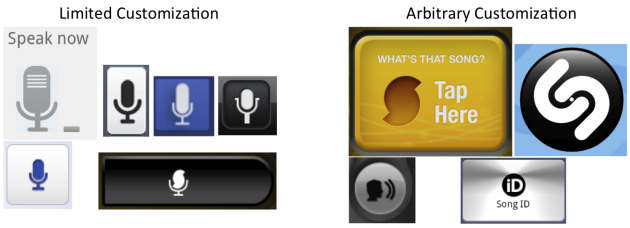

Figure 7: **Customization in Android UIs.** *Examples of UI elements corresponding to microphone access. The ones on the left could be accommodated with a standard ACG that offers limited customization options (e.g., color). The ones on the right require significant customization, or a change to the application's UI.*

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 4.1.1). Other systems, such as the EROS Trusted Window System (EWS) [31], 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 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 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 4.1.1, 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 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 7 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 4.1.1).

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 took 0.47 ms on average (standard deviation 0.22 ms).

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 [39]. We presented state-of-the-art permission systems on modern client platforms, such as iOS, Android, and browsers, in Section 2. We address other related systems here.

Browser restrictions have led web developers to use browser plugins and extensions to access user-owned resources. For instance, developers have created copy-and-paste buttons by overlaying transparent Flash elements and clickjacking [25]. Flash recently introduced a user-initiated action requirement [1], restricting paste to the Ctrl-V shortcut and requiring a click or keystroke for other permissions. This requirement does not accurately capture user intent, as users often perform actions unrelated to the access granted.

CapDesk [22] and Polaris [33] are experimental capability-based desktop systems that apply a philosophy similar to that of user-driven access control, but their focus is restricted to file access. Both give applications minimal privileges, but allow users to grant applications permission to access individual files via a powerbox (similar to our content picking ACG). Recent systems like Mac OS X Lion [3] and Windows 8 [20] have adopted this approach for file picking. We generalize user-driven access control for *all* user-owned resources.

Shirley and Evans [32] 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 [18] tries to infer the authenticity of browser-based file downloads using similar techniques. Our technique requires no such inference and robustly captures user intent at the moment a user interacts with an ACG.

The EROS Trusted Window System (EWS) [31], the Qubes OS [29], and Tahoma [8] advocate user-authentic gestures for copy-and-paste. NitPicker [14] and EWS support drag-and-drop. We generalize these notions into generic support for permission-granting input sequences. None of these systems consider the notion of ACGs, though EWS provides a special kind of transparent window that allows applications to include customized copy-and-paste buttons. We found in our evaluation that the transparent window solution enables attacks that our ACG design eliminates. None of these systems consider access to other user-owned resources.

Felt et al. uncovered permission re-delegation vulnerabilities [13] in Android by which applications unwittingly delegate permissions to unprivileged applications via IPC. Such threats may still exist on ACG-capable systems, but can be prevented by the IPC inspection mechanism proposed in [13].

Multi-level security systems like SELinux [26] classify information and users into sensitivity levels to support mandatory access control security policies. Information flow control techniques (e.g., [40]) can provide OS-level enforcement of information flow policies; user-driven access control can capture such policies from user actions.

8 Conclusion

In this paper, we advocate the use of *user-driven access control* for granting permissions 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 client platforms adopt user-driven access control.

9 Acknowledgements

We thank Stuart Schechter, Steve Gribble, Galen Hunt, Dan Simon, John Sheehan, and the UW CSE systems seminar for valuable feedback on this work, and Greg Akselrod, Roxana Geambasu, and Dan Halperin for feedback on earlier drafts. We thank Michael Berg and Microsoft User Research, Andrew Begel, James Fogarty, and Batya Friedman for their help with the user study, and the study participants for their participation. This work was done while the first two authors were visiting Microsoft Research.

References

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

[2] ANDROID OS. http://www.android.com/.

[3] APPLE. App sandbox and the Mac app store. https://developer.apple.com/videos/wwdc/2011/ Nov. 2011.

[4] APPLE. iOS4, 2011. http://www.apple.com/iphone/.

[5] BALLANO, M. Android Threats Getting Steamy. Symantec Official Blog, Febuary 2011. http://www.symantec.com/connect/blogs/android-threats-getting-steamy.

[6] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting Browsers from Extension Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (Feb. 2010).

[7] CHROMIUM. Security Issues. https://code.google.com/p/chromium/issues/list?q=label:Security, Feb. 2011.

[8] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy* (2006).

[9] DOWDELL, JOHN. Clipboard pollution. http://blogs.adobe.com/jd/2008/08/clipboard_pollution.html, 2008.

[10] FACEBOOK. Apps on Facebook.com, 2011. http://developers.facebook.com/docs/guides/.

[11] FELT, A. P., GREENWOOD, K., AND WAGNER, D. The effectiveness of application permissions. In *USENIX WebApps* (June 2011).

[12] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. Tech. Rep. UCB/EECS-2012-26, UC Berkeley, 2012.

[13] FELT, A. P., .WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium* (2011).

[14] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC* (2005).

[15] GRIER, C., TANG, S., AND KING, S. T. Secure Web Browsing with the OP Web Browser. In *IEEE Symp. on Security & Privacy* (2008).

[16] HOWELL, J., AND SCHECHTER, S. What You See Is What They Get: Protecting Users from Unwanted Use of Microphones, Camera, and Other Sensors. In *Web 2.0 Security and Privacy Workshop* (2010).

[17] KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., AND KAHN, C. E. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering 17*, 11 (Nov. 1991).

[18] LU, L., YEGNESWARAN, V., PORRAS, P., AND LEE, W. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACM CCS* (2010).

[19] MACKENZIE, I. S. Fitts' Law as a Research and Design Tool in Human-Computer Interaction. *Human-Computer Interaction (HCI) 7(1)* (1992), 91–139.

[20] MICROSOFT. Accessing files with file pickers. http://msdn.microsoft.com/en-us/library/windows/apps/hh465174.aspx.

[21] MICROSOFT. What is User Account Control?, 2011. http://windows.microsoft.com/en-US/windows-vista/What-is-User-Account-Control.

[22] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[23] MOTIEE, S., HAWKEY, K., AND BEZNOSOV, K. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In *Symposium on Usable Privacy & Security* (2010).

[24] MOZILLA FOUNDATION. Known Vulnerabilities in Mozilla Products, February 2011. http://www.mozilla.org/security/known-vulnerabilities/.

[25] NOVAK, B. Accessing the System Clipboard with JavaScript: A Holy Grail? http://brooknovak.wordpress.com/2009/07/28/accessing-the-system-clipboard-with-javascript/.

[26] NSA CENTRAL SECURITY SERVICE. Security-Enhanced Linux. http://www.nsa.gov/research/selinux/, Jan. 2009.

[27] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *ACM CCS* (2007).

[28] RUDERMAN, J. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2011.

[29] RUTKOWSKA, J., AND WOJTCZUK, R. Qubes OS. Invisible Things Lab. http://qubes-os.org.

[30] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SOSP* (2007).

[31] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS Trusted Window System. In *USENIX Security Symposium* (2004).

[32] SHIRLEY, J., AND EVANS, D. The User is Not the Enemy: Fighting Malware by Tracking User Intentions. In *New Security Paradigms Workshop* (2008).

[33] STIEGLER, M., KARP, A. H., YEE, K.-P., CLOSE, T., AND MILLER, M. S. Polaris: Virus-Safe Computing for Windows XP. *Communications of the ACM 49* (Sept. 2006), 83–88.

[34] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *USENIX OSDI* (2010).

[35] W3C. Device APIs and Policy Working Group, 2011. http://www.w3.org/2009/dap/.

[36] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium* (2009).

[37] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of Desktop and Web Applications on a Multi-Service OS. In *USENIX Hot Topics in Security* (2009).

[38] YEE, K.-P. User interaction design for secure systems. In *4th Conference on Information and Communications Security* (2002).

[39] YEE, K.-P. Aligning Security and Usability. *IEEE Security and Privacy 2(5)* (Sept. 2004), 48–55.

[40] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *USENIX OSDI* (2006).