User-Driven Access Control:
A New Model for Granting Permissions in Modern Operating Systems

Franziska Roesner*

Qualifying Examination Project — April 19, 2011
Advised by Tadayoshi Kohno (UW) and Helen Wang (MSR)

Abstract

Modern client platforms, such as iOS, Android, 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 the user's data that resides with various 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 propose *user-driven access control*, whereby permission-granting is built *into* existing user actions, rather than added as an afterthought via manifests or prompts. To this end, we introduce two OS-level techniques, *access control gadgets* and *kernel-recognized gestures*, for controlling access to user-owned resources. Our prototyping and evaluation experience indicates that user-driven access control is a promising solution for the problem of granting permissions for user-owned resources 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 [3] isolate applications into separate processes with different user IDs, and all web browsers implement the same-origin policy [32], which isolates one web site (or application) from another. By default, these principals receive limited privileges; they cannot, for example, access arbitrary devices or write to 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 fail to provide adequate functionality and security for access to the user's data and resources. From a functionality standpoint, isolation inhibits the client-side manipulation of user data across arbitrary applications. For example, the isolation of web sites makes it difficult to share photos between two online providers (e.g. Picasa and Flickr) without manually downloading and re-uploading them. From a security standpoint, existing access control mechanisms tend to be coarse-grained, abrasive, or inadequate. For instance, they require users to make uninformed decisions at install time via manifests [2, 4], or they unintelligently prompt users to determine their intent [3, 21].

Thus, a pressing open problem in these systems is how to allow users to grant applications access to *user-owned resources*, such as privacy-and cost-sensitive devices (such as the phone or camera), system services (such as the contact list or clipboard), and user content (such as photos or documents) stored with various applications. To address this problem, we propose *user-driven access control*, whereby the system infers user intent via authentic, natural user actions.

In particular, we introduce two operating system techniques to infer user intent: (1) *access control gadgets* and (2) *kernel-recognized gestures*. As with web mashups, access control gadgets allow developers to easily integrate kernel-level access control into an application context, while still allowing the kernel to infer authentic user intent. Kernel-recognized gestures bind certain keyboard, mouse, or touchscreen gestures to permission-granting events and standardize them across applications.

*In collaboration with Tadayoshi Kohno (UW), Alexander Moshchuk, Bryan Parno, and Helen Wang (MSR). Work done in part while employed by Collabera as a contractor for Microsoft Research.

Together, these techniques support **generic** access-control metaphors such as copy-and-paste, drag-and-drop, global search, file or content picking, device access, and long-term relationships among applications. They **minimize unintended access** by granting limited permissions to applications only when requested by the user, and they place a **minimal burden on users** by extracting a user's access-control intentions from his or her current actions and tasks (rather than, e.g., via prompts).

We built a prototype of user-driven access control into a system that isolates applications based on the same-origin policy. We use our prototype to evaluate access control gadgets and kernel-recognized gestures with respect to the functionality provided to users, as well as the impact of these techniques on existing applications. We find that kernel support for user-driven access control imposes nearly no performance overhead; e.g., it adds only 0.02 ms of delay to drag-and-drop events compared to Windows. A quantitative security evaluation shows that our system prevents a large swath of user-resource vulnerabilities. Finally, we include targeted user study data to support our design decisions.

In summary, we make the following contributions:

- We introduce the concept of *user-driven access control* for systems with strongly isolated least-privilege applications, whereby the intent embedded in user actions is translated into system-level access control decisions.
- We propose *access control gadgets*, which enhance application customizability while accurately capturing user intent, and we establish *kernel-recognized gestures* as a first-class access-control primitive.
- We demonstrate that user-driven access control can be added to a system in a manner that provides good performance, security, and extensibility.

2 State of the Art in Permission Granting

In this section, we motivate our work by identifying shortcomings in existing systems.

Global Resources. Traditional desktop systems expose user-owned resources to applications simply by globalizing them. Similarly, smartphone operating systems 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 study (Section 6.5) indicates that such exposures contradict user mental models (e.g., users expect data on the clipboard to remain private until pasted).

Manifests. Platforms like Android [2] and Facebook [10] use install-time manifests to allow applications to request permissions to access 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 without explicit user consent. Furthermore, studies indicate that many applications ask for more permissions than needed [5, 11], and recent Android malware outbreaks suggest that users still install applications that ask for excessive permissions [4].

Prompts. By contrast, iOS [3] prompts users the first time an application wishes to access a resource. Similarly, Windows displays a User Account Control prompt [21] when an application requires additional privileges to alter the user's system. 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, so users learn to ignore them [24, 45].

No Access. Some systems simply do not support application access to user-owned resources. For example, today's web applications cannot access a user's local devices except through browser plugins (which have access to all user-owned resources, thereby violating least privilege). While various efforts (such as HTML5) aim to allow web applications to access devices directly [39, 41], they have not yet specified how user actions should be mapped to access-control decisions. Research browsers [7, 13, 42] as well as browser operating systems [38, 43] also have not addressed access control for user-owned resources.

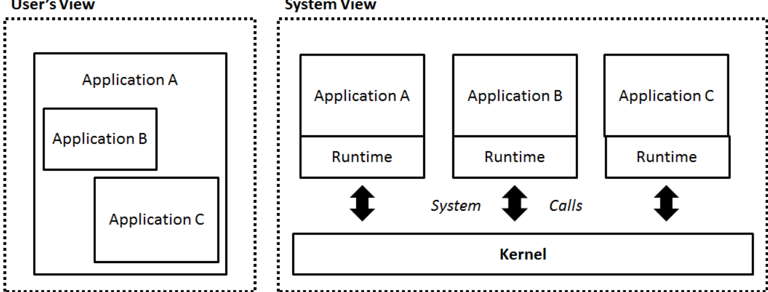

Figure 1: **Application Model.** *As in web mashups, applications may embed other applications, as shown in this example. Each application is isolated and runs atop a generic runtime (e.g., a browser renderer or Win32 library).*

3 Problem Definition and Goals

We consider the problem of allowing a system (Section 3.1) to accurately capture a user's access control decisions for user-owned resources (Section 3.2). Any solution in this space must **minimize the burden on users** while also **minimizing unintended access**. Section 3.3 presents the threats that may result in unintended access.

3.1 System Model

We assume that applications are isolated from each other according to some principal definition (such as same-origin policy), share no resources, and have no access to user-owned resources by default (see Figure 1). Applications may, however, communicate via IPC channels. 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 search engine may embed an advertisement. Like all applications, embedded principals are isolated from one another and from the outer embedding principal; this is unlike commercial browsers today, where all principals embedded on one web page share the same OS-level process [31]. 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) [36, 42].

To provide access control for user-owned resources, a system must support both access control *mechanisms* and access control *policies*. For the latter, 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.

3.2 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, and disk space. We consider all other resources to be user-owned, including:

1. Privacy- or cost-sensitive devices, both physical (such as the phone, the microphone, the GPS, and the printer) and virtual (such as the clipboard, the form autocomplete data store, the contact list, preferences, and the "transient clipboard" where data is stored during a drag-and-drop operation).
2. User-controlled capabilities or settings, such as wiping or rebooting the device.
3. Content, such as photos or documents, that resides in various applications.

3.3 Threat Model

We consider the attacker to be a malicious or compromised application, but we assume that the kernel is trustworthy and uncompromised; hardening the kernel is an orthogonal problem that has received considerable attention

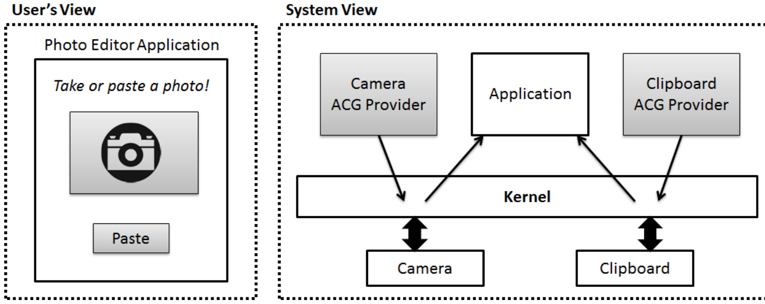

Figure 2: **Access Control Gadget (ACG) Overview.** *An application may embed ACGs to allow users to grant it resource access (to the camera and the clipboard, in this example). The ACGs are owned by their respective ACG providers, which instruct the kernel to grant the embedding applications access.*

(e.g., [28, 35, 44]). 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 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 an orthogonal problem to the problem of user-driven access control. Second, we do not consider the class of attacks on "what" data is accessed. Input sanitization to protect against malicious content is a separate research problem. Furthermore, since we assume that applications have full network and IPC access, an application that already possesses user data can always choose to leak it. Techniques like information flow control may help remove this assumption [17, 46], but again is orthogonal to our investigations.

4 Design: User-Driven Access Control

We investigate user-driven access control for user-owned resources in systems that strongly isolate applications. In particular, we develop two OS-level mechanisms that capture the user intent inherent in existing user actions. Access control gadgets (Section 4.1) allow applications to embed access control in an application context, while preserving the kernel's ability to authentically determine user intent. To provide shortcuts for common actions, we support kernel-recognized gestures (Section 4.2). These two techniques combine to support a broad range of natural metaphors by which users indicate access control decisions. These include drag-and-drop, copy-and-paste, global search, file or content picking, and device access.

4.1 Access Control Gadgets

At a high-level, an access-control gadget (ACG) is provided by an ACG provider, a principal trusted to exclusively mediate access to a particular user-owned resource (e.g., the camera). The provider exposes one or more *gadgets* that other applications can *embed* in their UI, similar to a web mashup (see Figure 2). For example, the clipboard provider may expose gadgets for cut, copy, and paste — we will return to more complex examples of ACGs in Section 4.1.4. The kernel enforces display isolation between embedded applications, and ensures that ACGs only receive authentic user inputs.

Based on the user's interactions with the ACG UI, the ACG provider instructs the kernel to grant the embedding application access to the resource via the appropriate mechanism (i.e., updating the kernel-level access

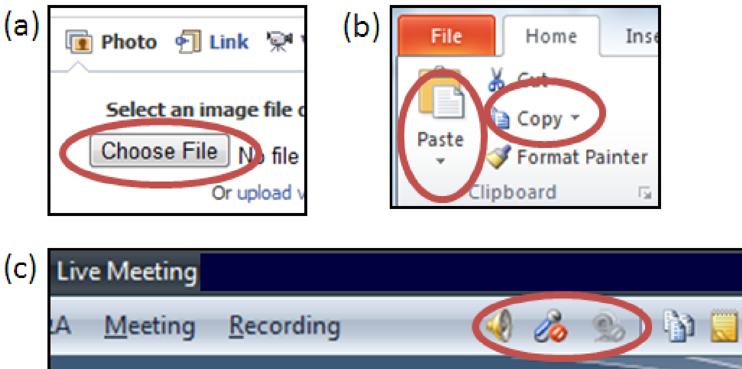

Figure 3: **Example Access Control Gadgets.** *ACGs apply to a broad range of user-driven access control, including (a) content picking, (b) copy-and-paste, and (c) device access.*

control list or directly mediating content transfer — see Section 4.3 for more details). The kernel enforces a number of properties to ensure the authenticity of the interactions between the user and the ACG (Section 4.1.2). Our design ensures the robustness of ACGs and adheres to the principle of least privilege (Section 4.1.3).

ACG providers are not device drivers (which reside in the kernel), but rather sit at a higher level of abstraction. For example, there is only one abstract camera resource, even if the system supports many hardware cameras.

4.1.1 Design Rationale

In designing ACGs, we aim to distill the advantages (and avoid the disadvantages) of two extreme points in the design space. At one extreme, the OS could permanently reserve a portion of the UI (e.g., a bar at the top of the screen) for access control decisions. This prevents malicious applications from manipulating the UI, but it limits how a legitimate app can customize the access control experience. Users are accustomed to making access-control decisions within an application context in ways that cannot easily be replicated by a fixed kernel UI. For example, many programs allow users to paste via context menus or toolbar buttons; indeed, a study of Microsoft Office data indicates the paste button is the most clicked button on the toolbar [14], despite several other more efficient options for pasting. Finally, particularly on newer client platforms like tablets and smartphones, screen real estate is at a premium; dedicating this resource to infrequent access-control decisions is wasteful.

At another extreme, the application could request an OS prompt whenever an access control decision is needed, so that, most of the time, the full UI would be available for application usage. However, prompts limit customizability, disrupt the user's interactions with the application, and have all of the shortcomings we discussed in Section 2.

Our design of ACGs provides the authenticity of dedicated real estate without monopolizing the display. ACGs integrate into the application, rather than disrupting it as prompts do. In Section 4.2, we introduce kernel-recognized gestures, which share many of these benefits and act as shortcuts for some popular ACGs. However, kernel-recognized gestures potentially limit applications by restricting the "palette" of gestures the application can employ, so we aim to use them sparingly. They also do not offer the full expressiveness of an ACG.

4.1.2 Ensuring Authentic Interactions with ACGs

To accurately convert the intent of user actions into OS-level access control decisions, the kernel must ensure the integrity of user interactions with ACGs. In particular, if we model the user's basic interactions with the ACG as the following "protocol":

$ACG \rightarrow User$: Purpose (e.g., toggle camera access)
$User \rightarrow ACG$: Intent (e.g., please grant camera access)
$ACG \rightarrow User$: Status and/or revocation option

then the kernel must ensure the integrity of each "message" and authenticate the source of *User* messages; we discuss these protections below. Helping the user distinguish fake ACGs from real ACGs is less critical; a fake ACG cannot grant the embedding application access to any additional resources. However, applications that

embed other applications complicate this picture, as it potentially makes it more difficult for the user to distinguish which app will be authorized by interactions with an ACG. We discuss this complication below.

Protecting ACG Purpose Indicators. To provide integrity for the ACG's "Purpose message" (e.g., displaying a "Copy" button), the kernel only activates an ACG when the entire ACG is visible (i.e., at the top of the display's Z ordering), since any overlays could manipulate the ACG's message. For example, a malicious application might try to overlay its own labels to reverse the meaning of an ACG's copy and paste buttons. For the user to accurately receive the ACG's message, we must also ensure the user has time to perceive it. Thus, the kernel only activates an ACG after it has been fully visible for at least 200 ms (we use a fade-in animation to convey the activation to the user). This delay gives the user sufficient time to react [18], and we postulate (as do the developers of Chrome [6] (Issue 52868) and Firefox [25] (Advisory 2008-08)) that it does not inconvenience the user in normal circumstances. The ACG is disabled (and greyed out) if any portion becomes obscured.

Protecting User Intent Indicators. To protect the user's "Intent messages" (e.g., mouse clicks), the kernel transmits to ACGs only input events that originated from physical devices (e.g., keyboard, touchscreen, or mouse). Other software cannot generate these user-initiated messages. 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 displayed when a user hovers over a gadget. This contributes to the previous property (ensuring gadgets are at the top of the visual stack), and prevents a malicious application from confusing the user about where they are clicking within an ACG.

Protecting Status and Revocation Options. Finally, the user should be able to assess the status of the access control provided by an ACG (e.g., determine that camera access is enabled) and indicate a desire to revoke that access (e.g., terminate an application's ability to take pictures). To support this, the kernel provides an ACG "Control Panel", i.e., a trusted application that can always be accessed via a secure attention sequence (e.g., Ctrl-Alt-Delete). The ACG Control Panel allows the user to survey the state of the system's ACGs and access revocation functions. Prior work suggests mechanisms for intuitively conveying access status [15, 30].

Nested Principals. Care must be taken to accurately capture user intent when an application includes embedded principals (e.g., a web search engine which embeds ads), which may be arbitrarily nested (e.g., a mashup page which embeds a web search engine which embeds ads). In particular, the kernel must prevent an embedded application from tricking users into believing that the gadget it owns is actually owned by the outer embedding application. For example, a malicious ad embedded in `google.com` might embed an ACG for GPS access. If the ad customizes its UI to mimic Google, the user may be tricked into thinking the GPS gadget will grant geolocation access to Google, rather than the ad.

To prevent such confusion, only a "top-level" or outermost application may embed ACGs by default, though it may choose to grant the applications it embeds the permission to themselves embed ACGs. While we do not attempt to solve the principal identification problem (see Section 3.3), we postulate that of all the various principals on the system, the user is most likely to understand and identify top-level principals, rather than principals that may be nested arbitrarily deeply in other applications. Furthermore, compared with the OS, the embedding application has at least as much information (and typically more information) about the principals it embeds. For example, Google can distinguish embedded ads that should not receive geolocation access and embedded services, such as Google Maps, that should.

4.1.3 Keeping ACGs Robust

Since ACGs are trusted entities, we design them to have minimal privileges and to be robust against the consequences of a coding, design, or UI error.

Source. For simplicity, we currently require ACG providers to come pre-configured with the operating system; they are updated via the same update mechanism used for OS updates. An application can query the system to learn about available ACGs and their capabilities, similar to how COM objects are discovered today. However, unlike COM, applications cannot request that a new ACG be installed.

Design Philosophy. ACGs should be kept simple and limited; like early Unix programs, they should do one thing and do it well. If an ACG provides any customizability (generally discouraged), it should be extremely limited.

For example, rather than allowing an application to specify an arbitrary font size, an ACG might allow the app to choose from a few possible font sizes in the system's default language.

Furthermore, each ACG provider operates as a distinct OS principal and manages a single user-owned resource. This provides defense in depth and respects the system's spirit of least privilege. A compromise of one gadget provider will not affect the security of any other resource.

Restrictions. Finally, to limit opportunities for compromise, ACGs are permitted neither network access nor IPC access. An ACG and its embedding application need not communicate directly: if the ACG provides any customization options, the application can select amongst them when it creates the embedding. Software updates of the ACG included with OS updates.

4.1.4 Example ACGs

In this section, we describe several examples of how ACGs apply to various user-driven access-control scenarios. As depicted in Figure 3, ACGs can be used to grant an application access to arbitrary user-owned resources. This access may be granted for different access durations (one-time, session-based, permanent), though some devices may not support all durations (e.g., the printer and the clipboard support only one-time access). We discuss duration in more detail in Section 4.1.5.

Device Access. To request access to a device like the camera or the printer, an application may embed camera or printer ACGs. When a user clicks on one of these ACGs, the application either receives content (e.g., from the camera) or is prompted for content (e.g., to send to the printer). Similarly, an application may use ACGs to request access to any device supported by the system.

Content Picking. As a more complex example, we describe our design for a content-picking (a generalization of "file picking") ACG provider that lets users browse cross-application content. Within the ACG, the user selects one or more content items; the embedding application is then granted access to that content.

The content-picking provider exports two gadgets: **(1)** A gadget similar to the one shown in Figure 3(a) to initiate content picking, and **(2)** A gadget that provides an interface for viewing and selecting content.

As shown in Figure 4, the second gadget allows the user to navigate content across applications. The content-picking provider does not gain access to the user's content across applications. Rather, when the user initiates content picking, the content-picking provider, via the kernel, prompts the queried applications to return *content views* the second content-pick gadget embeds. When the user selects a content view, the embedding application is granted access to that content.

Like all embedded principals, the content view is isolated from the principal that embeds it (in this case the content-picking ACG). We chose this design for two reasons: **(1)** The content in the content view comes from arbitrary applications and is not trusted. By isolating the content view, we protect the trusted content-pick ACG. **(2)** Allowing each application to control its own content views allows it to intelligently control user interaction with the content based on its semantics (e.g., Facebook friends are a specialized kind of content that would lose richness if handled generically).

User Data Search. Rather than picking specific content, a user may wish to search his or her applications. We provide this functionality via an ACG that implements the equivalent of cross-application desktop search.

Similar to the content-picking example, the search ACG exports two gadgets an application can embed. The first triggers the main search gadget. When triggered, the search gadget allows the user to select a subset of applications. Via the kernel, the ACG submits the user's query to these applications, which define search according to their own semantics, including whether the content search is local, remote, or both. As before, the search gadget is not exposed to the search results themselves, which are displayed via content views returned (along with a relative ranking) by the applications. As with traditional desktop search, the search gadget must interleave the returned results appropriately and display them to the user. Future work must consider how the search gadget should rank results received from different and potentially untrusted applications.

When the user selects a result, the application that embedded the search gadget is granted access to that result. This process is identical to the content picking process depicted in Figure 4.

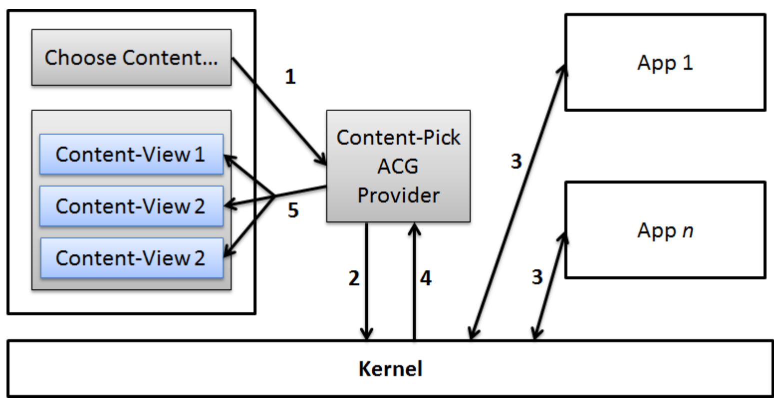

Figure 4: **Content Picking via ACGs.** *An application may embed a content-picking ACG. When the user (1) interacts with the ACG, the ACG provider (2) prompts the kernel to (3) query the selected applications for content. The queried applications return isolated content views, which are (4) passed to the ACG provider, which (5) embeds them in the content-picking ACG. The user may select content from one of these views, thereby enabling the original embedding application to access it.*

4.1.5 Duration of Access Control Decisions

An ACG may specify that a particular permissionshould be one-time, session-based, or permanent; see Section 4.3 for more detail. Session-based access is revoked when the application is closed. Both session-based and permanent access may be revoked by the user via the appropriate ACG. Recall from Section 4.1.3 that the kernel ensures the user can always access the system's ACGs, even if the original application is closed or refuses to cooperate.

It is the ACG provider's responsibility to decide how to expose these options (if at all – devices like the printer and the clipboard may support only one-time access). For example, the camera ACG might display three variants of a camera button to allow the user to indicate whether to provide one-time access (e.g., a single photo), session-based access (e.g., a video stream for a Skype call), or permanent access (e.g., to photo editing software).

4.2 Kernel-Recognized Gestures

4.2.1 Overview

To increase the usability of a variety of common access control decisions, we introduce the use of authentic shortcuts via kernel-recognized gestures. Kernel-recognized gestures bind certain keyboard, mouse, or touchscreen gestures to permission-granting events and standardize them across applications. In other words, the kernel monitors the stream of inputs from the user. When it recognizes a particular gesture or sequence of inputs (e.g., Ctrl-C), it interprets that gesture as granting access permissions to the in-focus application (e.g., granting it the ability to write data to the clipboard).

As we discuss in Section 7, while prior work has suggested specific reserved gestures [12, 33, 36], we generalize this notion and make it a first-class primitive for supporting user-driven access control. Note that kernel-recognized gestures are similar to, but distinct from, secure attentions sequences like Ctrl-Alt-Del [16]. Secure attention sequences are also kernel-recognized and intercepted gestures, but they allow the user to bring up an authentic kernel-owned UI (usually a login prompt). By contrast, our kernel-recognized gestures are used by the kernel to interpret genuine user intent and are mapped to system-level access control decisions for applications.

4.2.2 Usage

Kernel-recognized gestures allow the system to support shortcut-based copy-and-paste as well as the drag-and-drop gesture, both common access control idioms in desktop operating systems. In particular, a user may use the reserved Ctrl-X, Ctrl-C, and Ctrl-V shortcuts to permit data access between applications via the clipboard. Similarly, a user may use the reserved drag-and-drop gestures to permit data access between applications via the transient clipboard. The system also supports other shortcuts to allow users to grant various permissions to applications, including Ctrl-P for print or PrtScn for a screenshot.

To assess whether kernel ownership of these gestures would unduly restrict applications, we performed an analysis of the top 100 unique[1] Windows applications based on data from July to August 2010. We found that 100% of these applications support standard copy-and-paste gestures (Ctrl-X, Ctrl-C, Ctrl-V) or simply do not support copy-and-paste (e.g., the Java runtime). This speaks strongly to the de facto standardization of these gestures.

4.2.3 Tradeoffs

Benefits. Kernel-recognized gestures have several benefits. First, they are easy for the OS to interpret; the OS is already responsible for collecting user input in its role as the device manager, and it must also track the in-focus application in its role as display manager. Thus, it is straightforward to add functionality to detect particular inputs and apply permission changes to the in-focus application. Gesture standardization is also beneficial to users, who do not need to learn or recall separate gestures for every application. Particularly for expert users, these shortcuts are likely to be more efficient than interaction with ACGs. Finally, unlike ACGs, kernel-recognized gestures do not require any dedicated display real estate.

Limitations. A clear drawback of reserved kernel-recognized gestures is the lack of customizability; neither applications nor users may be permitted to customize these shortcuts, for fear that a malicious app or a careless user may change the significance of the gestures (e.g., change Ctrl-C to indicate paste). Of course, the kernel can still forward the relevant inputs to the application, allowing the application to apply its own internal interpretation. For example, a video game might decide that when the user hits Ctrl-C, it indicates the game character should run left; the game simply ignores the access it receives to the clipboard (our implementation does not permit applications to "horde" access tokens; an application can only take advantage of access granted to it or let the opportunity pass on by). Applications can also implement additional gestures internally; e.g., an application could still use the sequence "yy" to indicate copy internally. However, the data copied would not be placed on the system clipboard unless the user employed the appropriate kernel-recognized gesture or corresponding ACG.

Nonetheless, due to these limitations, we restrict our usage of kernel-recognized gestures to the most universal gestures. Less standardized access control patterns can still be supported via ACGs.

4.3 Access Mechanisms

Once the user has granted an application access to a resource via an ACG or a dedicated gesture, the kernel must take a system-level action to enable the access. Here the kernel makes a distinction between *one-time* and *longer-term* (session-based or permanent) access. We describe this distinction here.

In the case of longer-term access, the kernel simply updates an access control list (ACL) to grant the appropriate application access to the APIs associated with the relevant resource. For example, when a user grants an application session-based access to the camera, the kernel updates its ACL to permit the application to directly access the camera's APIs (e.g., `TakePhoto()`). This model aligns with user expectations: after receiving long-term access, an application should be able to access the relevant APIs at any time until access is revoked.

For one-time access, a user's expectation is more specific with respect to the moment in time when the permission may be exercised. That is, when the user grants an application access to a resource, he or she expects that access to happen precisely at that moment. For example, when the user grants an application one-time access to the camera, the application should not be able to defer its access until thirty seconds later, at which point the camera may no longer be directed at the scene intended by the user. Thus, updating the ACL for one-time access is insufficient; it must be associated with a timeout that prevents delayed use of the permission.

Because determining an appropriate timeout is extremely difficult in a non-real-time system with arbitrary delays, we take another approach. For one-time access, the kernel acts as a mediator of content. At the moment of the user's permission-granting action, the kernel pulls content from the associated resource and pushes it to the appropriate application. We discuss the mechanics of this process further in Section 5 below. Contrary to the longer-term case, the application does not receive direct access to any APIs associated with the resource, and any delays during this process (e.g., time required for the camera to turn on) cannot be exploited by the receiving application.

[1] That is, we counted different versions as the same application.

Type	Call Name	Description
syscall	EmbedACG(location, resource, type, duration)	Embeds an ACG in the calling application's UI
syscall	GrantAccess(src, dest, type, duration)	Instructs the kernel to permit access from dest to src principal
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

Figure 5: **User-Driven Access Control System Calls and Upcalls.** *Note that the `GrantAccess()` system call may be issued only by a registered ACG provider and only affects the resource for which it is responsible. The windowId allows a multi-window application to determine which window should respond to the upcall.*

5 Implementation

In this section we describe our implementation of ACGs and kernel-recognized gestures as part of a complete system that strongly isolates applications.

5.1 Baseline System

We build our new mechanisms into an anonymized baseline system [23] described in this section. This system runs on Windows and is implemented as a lightweight software layer between the kernel and applications.

Our system isolates applications using Drawbridge [29], a sandboxing technology that uses a "library OS" approach to virtualize host resources and push complicated Windows kernel-mode components into user space. Isolated applications interface with the host OS via a narrow API of around 30 calls (compared to over 100,000 in Windows), which support allocating virtual memory, threading, synchronization, and generic host-mediated I/O.

Our system's kernel works with the Drawbridge security monitor [29] to enforce an isolation policy that labels each application with its remote origin, as defined by the same-origin policy [32]. For example, a web application `http://www.cnn.com/us/news/` corresponds to the principal `http://www.cnn.com`. The system runs both web and desktop Win32 applications. It extends the Drawbridge API with a dozen more calls to support features such as embedding content rendered by other principals, downloading HTTP/HTTPS content, and performing cross-principal messaging.

Isolated applications expose their display to a custom UI shell via remote desktop protocol (RDP) messages. The kernel manages the position, dimensions, z-index, and transparency of each application's windows consistent with our design requirements in Section 3.1.

5.2 Overview

In our implementation, we extend the system described above with about 700 lines of C# code to allow the kernel to **(1)** determine user intent via gadgets or gestures, and **(2)** act on that intent by taking the appropriate access control actions. Figure 5 summarizes the system calls and application upcalls we implemented to support user-driven access control. Note that only the kernel may issue upcalls.

When a user indicates an intent to grant an application access to a user-owned resource, he does this either through interaction with an ACG or with a kernel-recognized gesture. We discuss our implementation of both methods of intent detection in Section 5.3.

Once the kernel has determined the user's intent, it must make the appropriate access control changes (Section 5.4). The appropriate changes depend on the semantics of the resource and the type of access granted.

5.3 Determining User Intent

We discuss the implementation of gadgets and gestures, which allow users to indicate an intent to grant access.

5.3.1 Supporting ACGs

In our system, access-control gadget providers are implemented as separate applications that provide access-control and UI logic and expose a set of ACGs.

The `EmbedACG()` system call allows other applications to embed access-control gadgets. Such 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 and the duration of access, depending on the set of gadgets exposed by the provider.

For instance, an application wishing to embed a copy button would make an `EmbedACG(`(x, y)`, "clipboard", "copy")` call during its UI setup, where "clipboard" is the resource and "copy" is the gadget type. Similarly, to embed a gadget that enables session-based GPS access, an application would call `EmbedACG(`(x, y)`, "gps", "toggle", "session")`.

Typically, we expect the various application runtimes to expose this call to applications. For example, to make it easier for web applications to adopt ACGs, we modified the browser runtime to expose this call as a new HTML tag, so that a web application can include, for example, `<acgadget src="clipboard://copy/">`.

The kernel binds these embedded regions to the appropriate registered ACG provider. The provider completely owns the UI stack within this region and can thus implement the gadget appropriately.

Once the ACG provider has determined the proper access control action via interactions between the user and the ACG, it either implements the action directly, or implements the action via the kernel. The provider implements the action directly for resource APIs that simply change state and do not require data transfer (e.g., muting the speaker). For all other forms of access, the provider triggers the kernel with a `GrantAccess()` system call, available only to registered ACG providers.

The `GrantAccess()` call specifies which principal should receive access (dest) to which other principal (src), along with the type of access granted, and the duration of the grant. For example, if the user clicks on a camera ACG in a photo editor to take a picture, the camera ACG provider makes a `GrantAccess(cameraId, photoEditorId, "photo", "one-time")` system call. Either the source or destination of a `GrantAccess()` call will always be the resource guarded by the ACG provider. The kernel enforces this restriction. We discuss the kernel's implementation of `GrantAccess()` calls below in Section 5.4.

5.3.2 Gesture Implementation

In our implementation, we focused on gestures for the most commonly used shortcuts in desktop systems: shortcuts accessing the clipboard (via cut, copy, and paste) and the transient clipboard (via drag-and-drop). These implementation details are representative of what is required for any shortcut (e.g., PrtScn or Ctrl-P).

We modified the kernel to route mouse and keyboard events to gesture detection logic that we added to the kernel; the events are then relayed to the application via RDP. This gesture detection logic required adding 120 lines of code to the kernel.

The kernel reserves the most common keyboard shortcuts for cut, copy, and paste: Ctrl-X, Ctrl-C, and Ctrl-V. The first two are effectively translated into the call `GrantAccess(inFocusApp, "clipboard", "put")`, while Ctrl-V becomes `GrantAccess("clipboard", inFocusApp, "get")`.

The kernel also implements drag-and-drop detection. It detects a drag by detecting a MouseDown followed by a MouseMove event on a draggable object (as determined by `IsDraggable()` calls to the application owning the object), and it detects the subsequent MouseUp event as a drop. Similar to cut, copy, and paste, a Drag is translated into the call `GrantAccess(inFocusApp, "transient clipboard", "put")`, and drop is translated into `GrantAccess("transient clipboard", inFocusApp, "get")`.

To support existing usage idioms for drag-and-drop, our implementation also dispatches `IntermediateEvent()` upcalls during a drag action. This allows our system to dispatch DragOver events while the mouse moves as well as DragEnter and DragLeave events when the mouse crosses window boundaries, allowing applications to show visual feedback as appropriate.

5.4 Acting on User Intent

When the kernel receives a `GrantAccess()` system call, either explicitly from an ACG provider or implicitly from a gesture, it may do one of two things, depending on the semantics of the type of access granted.

Set Access Control State. If the granted access is session-based or permanent, the kernel sets access control state indicating that the specified principal may directly access the APIs associated with the specified resource. Recall that users may use ACGs to revoke access as well; ACG providers notify the kernel of this via a `GrantAccess(principalId, null, "revoke")` call.

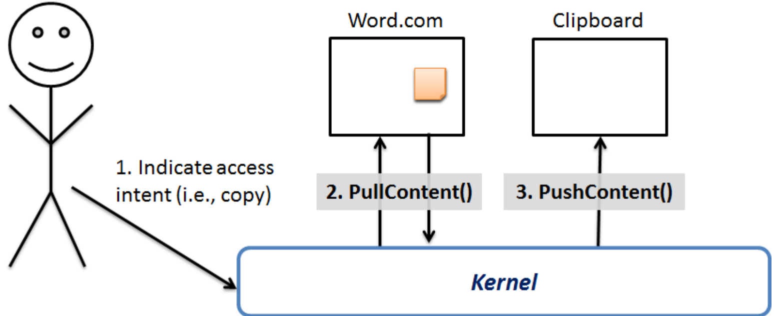

Figure 6: **Discrete Content Transfers.** *For one-time access grants, the user first* ***(1)*** *indicates an access-grant intent via an ACG or gesture. The kernel* ***(2)*** *issues a* `PullContent()` *upcall to the source application, which identifies and returns the relevant content. The kernel then* ***(3)*** *issues a* `PushContent()` *upcall to the destination application and passes it the content. In this example of a copy event, the source is a word processer and the destination is the clipboard.*

Discrete Content Transfer. Following `GrantAccess()`, if the granted access is one-time access, the kernel acts as a mediator for the transfer of one content item from the source to the destination. Here the kernel makes use of the upcalls in Figure 5. Specifically, the kernel issues a `PullContent()` call to the source, and then issues a `PushContent()` call to the destination (see Figure 6). For example, when the camera ACG provider grants access to a photo, the kernel issues a `PullContent()` upcall to the camera device, followed directly by a `PushContent()` upcall to the application embedding the ACG. As another example, granting access to the printer prompts the kernel to first `PullContent()` from the requesting application and then `PushContent()` to the printer. Note `PushContent()` merely pushes the specified data to the destination; it does not permit the destination read access to the source.

5.4.1 Application Support

Applications are responsible for handling push and pull upcalls appropriately. On a pull, this means that the application uses application-specific knowledge to determine which content is desired (e.g., the currently highlighted text or the most recently taken photo) and return it to the kernel. On a push, the application must inject the received object into the designated destination. We implemented this support for copy-and-paste and drag-and-drop in the browser renderer shared by web applications, adding about 300 lines of C# code.

6 Evaluation

Below, we evaluate our system on its extensibility, ease of use, security, and performance. We also present targeted user study data to evaluate a number of design decisions.

6.1 Extensibility

In this subsection, we evaluate the ease of extending our system to include access control support for a new user-owned resource.

In general, to support a new user-owned resource in the system, we: **(1)** add resource-specific support (system APIs and functionality) to the kernel if necessary, and **(2)** implement a gesture detector or an ACG provider and its associated gadgets. Here we discuss the implementation effort required to support several new resources; The first two columns in Figure 7 show an overview.

Search. To understand the complexity of supporting global search of a user's content across application, we have implemented a search ACG (Section 4.1.4).

For step 1, we modified the kernel API for the `GrantAccess()` call to allow the search ACG provider to specify a wildcard source indicating the set of running applications, rather than a single source (as described earlier). The kernel iterates through the applications, making `PullContent()` calls. It passes the resulting

	Approximate lines of C# code added to:		
	Kernel	**ACG Provider**	**Application**
Global Search	170	50	10
Copy-and-Paste	n/a	70	70
Drag-and-Drop	70	35	200
Camera Access	n/a	30	20

Figure 7: **Evaluation of Implementation Effort.** *For global search, ACG support does not include an algorithm to rank search results; it simply displays them. Similarly, application support for search refers only to the compilation and sending of search results, not full application-specific search. For the rest, application support is added to the runtime (a browser renderer) and simply exposes the relevant functionality to the web application.*

handles to content views from the applications back to the ACG provider via a `PushContent()` call. Adding this kernel support required 170 lines of C# code.

For step 2, implementing the search provider and its gadget was also simple: the gadget contains a text box along with a search button, requiring about 30 lines of code. The user's click on the search button prompts the search ACG provider to issue the appropriate `GrantAccess()` call to the kernel. In response to the upcall, queried applications in our prototype each return a set of content views along with relevance information. The kernel passes these to the search ACG, which reorders the results appropriately and displays them by embedding the content views. This took about 20 lines of code, since we only implement 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.

Clipboard and Transient Clipboard. Copy-and-paste requires ACG provider support, but it did not require any additional kernel support. Our implementation of the clipboard ACG provider required about 70 lines of code to invoke `GrantAccess()` and respond appropriately to `PullContent()` and `PushContent()` calls.

Support specific to drag-and-drop required about 70 lines of kernel code to provide the appropriate `Inter-mediateEvent()` calls. Since drag-and-drop is fundamentally only a gesture, the drag-and-drop ACG provider exposes no gadgets, it merely mediates access to the transient clipboard.

Camera. To assess the ease of enabling access to new physical devices, we added support for cameras. We leveraged existing system API support for the camera, so no kernel modifications were needed. The camera ACG provider, which exposes a gadget that allows a user to take a photo and immediately share this photo with the application embedding the gadget, required about 30 lines of code. When the user clicks on the gadget, the ACG provider must merely (1) issue a call to the camera to take a photo, and (2) issue a `GrantAccess()` call to the kernel to trigger a transfer of the photo to the receiving application.

6.2 Ease of Use

To evaluate the effort required to add new access-control functionality to applications, we studied the addition of several features to web applications. In many cases, we were able to provide most of the necessary support in the runtime (in this case a wrapper for the rendering engine of an existing commercial browser) shared by web applications.

The "Application" column in Figure 7 summarizes the implementation effort required to implement generic support in the runtime for search, basic copy-and-paste, drag-and-drop, and camera access. Note that the runtime simply exposes the relevant content to the underlying HTML; we do not evaluate the effort required to build a web application that makes use of this content in a meaningful way.

Rich File Sharing Case Study. To demonstrate the power of combining user-driven access control with existing web applications, we also implemented rich drag-and-drop file sharing across web applications. By adding proper support in the runtime, we were able to enable drag-and-drop file content across websites without any modifications to these websites.

More precisely, our implementation takes advantage of new features offered by HTML5 that dovetail nicely with our user-driven access control techniques. In particular, the HTML5 standard recently introduced support for a drag-and-drop API and a File API [40]. These new features allow websites to support the dragging and dropping

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	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)

Figure 8: **Relevant browser vulnerabilities.** *We categorize Chrome [6] and Firefox [25] vulnerabilities that affect user-owned resources; we show the percentage of vulnerabilities that user-driven access control eliminates by design.*

of files from a user's local file system to the web page (enabling drag-and-drop photo uploaders like DropMocks[2]). However, these mechanisms *do not support* the dragging and dropping of files across web applications (e.g., Facebook to DropMocks). In our system, we easily enabled this feature across web applications that support drag-and-drop with File API interaction.

Our wrapper interacts with the browser runtime to properly handle the `PullContent()` and `PushContent()` upcalls. Specifically, the wrapper:

1. on drag of an image, encapsulates the image in a file object and returns this file object to the `PullContent()` upcall.
2. on any drop, triggers the underlying HTML element's `ondrop()` handler with the content provided via the `PushContent()` upcall.

We note here that there are other ways in which web applications could take advantage of our system. For instance, rather than transferring large content on the client side via the File API, a source application might simply transfer an OAuth token [27] to the sink application. This would allow the sink application to access the specified data directly via a back-end API, without disclosing the user's credentials for the source application.

6.3 Security

To assess the security benefit of user-driven access control techniques, we evaluate their effectiveness against attacks on user-owned resources in browsers today.

We assembled a list of 631 publicly-known security vulnerabilities in recent versions of Chrome (2008 to 2011) [6] and Firefox (version 2.0-3.6) [25]. 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 [13, 42] targets this class of vulnerabilities.

Our focus is on the remaining vulnerabilities, which represent the future vulnerabilities that cross-principal isolation will not address. Of those remaining, the dominant category (with 30% in Chrome and 35% in Firefox) pertains to access control for user-owned resources among applications. We further sub-categorize these remaining vulnerabilities and analyze which ones can be eliminated by our user-driven access control. Figure 8 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). We identified 21 such vulnerabilities in Chrome and 18 in Firefox. With user-driven access control, most of these data leakage vulnerabilities can be eliminated, as only genuine user interaction with ACGs or kernel-recognized gestures grants access.

Nine of these vulnerabilities in Chrome are related to autocomplete functionality. While the use of an ACG for the autocomplete datastore can eliminate all but two of these bugs, the design of ACG support for autocomplete merits further consideration. In particular, it requires applications either to opt in to autocomplete (losing the application-agnostic benefits of autocomplete in browsers today) or for every textbox to be an embedded autocomplete ACG. The two vulnerabilities that we do not address by design are errors that could be duplicated in an ACG provider's implementation (e.g., autocompleting credit card information on a non-HTTPS page).

[2]http://www.dropmocks.com

Local Resource DoS. Eleven vulnerabilities (ten in Chrome and one in Firefox) allow malicious applications to perform denial-of-service attacks on a user's resources. For example, an attacker might download content without the user's consent. User-driven access control eliminates the attacker's ability to access user resources without user consent. Even with that consent, one-time access control gadgets and gestures grant only one access.

Clickjacking. We identified four clickjacking vulnerabilities in Chrome and one in Firefox. As discussed in Sections 4.1.2 and 4.1.3, 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 ACG provider. 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, for example, issuing clicks or forcing drag-and-drop actions. We identified three such vulnerabilities in Chrome and four in Firefox.

Cross-Application Exploits. We categorized a number of vulnerabilities as cross-application exploits, in which an attacker uses a user's transfer of content on the client side as an avenue for attacking another application (e.g., copying and pasting into vulnerable applications allows cross-site scripting). As described in our threat model, we consider the hardening of applications to malicious input to be an orthogonal problem. However, we do eliminate one such bug in Firefox: this bug involves a malicious search plugin a user may be tricked into installing, an attack eliminated by the search ACG provider in our design.

6.4 Performance

User-driven access control performance is fundamentally subject to the user's perception, as all support for granting access to user-owned resources begins and/or terminates with a UI event. We focus on investigating the performance of 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 user's perception of the usability of a system is particularly sensitive to performance degradation in drag-and-drop event handling, as it is a continuous, synchronous action.

In particular, we evaluate the performance of intermediate drag-and-drop events. These events (DragEnter, DragOver, DragLeave) are fired on every mouse move at some granularity while the user is dragging an object. To evaluate the performance of our system enhanced with user-driven access control, we compared to the performance 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 it took from the registration of the initial mouse event by the kernel to the triggering of the relevant intermediate drag event handler in the application. We found the difference to be 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). From personal experience using the system, we can also confirm qualitatively that the user experience of our prototype system is not impacted by any slowdowns.

6.5 User Study

To understand user mental models of cross-application access control and to evaluate a number of design decisions, we performed an online user survey (including questions and tasks) with 139 participants of varied backgrounds. Participants were recruited with the help of Microsoft's user research lab [22]. Self-reported demographic data indicates we had 111 males and 28 females, ages 18-72, with a wide variety of occupations, including custodian, teacher, and airline pilot. We present results from this study here, beginning with a general discussion of user mental models of desktop sharing abstractions, before discussing the design and results of three tasks.

Mental Models. 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) of users believe, wrongly, that applications can only access the global clipboard when the user pastes (see Figure 9). User-driven access control would match this mental model.

Participants had no consistent mental model of the security of drag-and-drop. The responses to the question about whether an application over which an item is dragged can access the item are statistically indistinguishable

Response Option	Responses
“A program will always see the text I’ve copied.”	11.51%
“A program could see the text, but probably wouldn’t look unless I hit paste.” (correct)	23.74%
“A program can only see the text if I pasted it into the program.”	40.29%
“Other programs can never see the text that I’ve copied.”	6.47%
“I don’t know.”	17.99%

Figure 9: **Responses to a multiple choice question asking when applications can access the clipboard.** *These results are statistically significant compared to an even distribution of responses ($\chi^2(N = 139) = 47.5827, p < 0.0001$).*

Figure 10: **Percentages of participant trials succeeded or failed during cursor/button attack scenarios.** *Participants were asked to copy and paste from one application to another; the categories on the x-axis indicate whether the task was to copy from application 1 to application 2 or vice versa. Application 2’s buttons randomly showed the wrong cursor text; the x-axis categories indicate whether copy, paste, or both (“attack”) buttons were wrong. Only scenarios in which attacks actually affected the required task are shown (e.g., a copy attack when not asked to copy from a malicious application would not have been encountered by participants).*

($\chi^2(N = 139) = 1.4820, p = 0.4766$) from an even distribution across the three possible responses (“true”, “false”, “don’t know”).

Trusted Cursor, Untrusted Buttons. In one of the tasks in our user study, we evaluated a design from related work. The EROS Trusted Window System (EWS) [36] supports application-customized copy-and-paste buttons via authentic transparent windows. While EROS controls the cursor when the user hovers over such a window, the application controls the visual appearance of the button. This allows for full application customizability at the expense of security guarantees: an application could register an area of the screen as a paste button but draw anything it wishes. This type of attack is impossible in our system.

To evaluate the importance of preventing such an attack, we presented our participants with a copy-and-paste task in which the cursor was trustworthy but the buttons were not. We found that most users (61%) reported not even noticing the additional information presented by the trusted cursor. Of those that noticed the cursor text, only 44% reported noticing the occasional inconsistencies between the cursor text and the button text. During the trials, users by and large believed the button text, not the cursor, failing in 87% of attack trials (see Figure 10 for more detail). Further, introducing the cursor text in the task instructions for half of the participants did *not* significantly affect either noticing the cursor ($\chi^2(N = 130) = 2.119, p = 0.1455$) or success rate ($F(1, 1) = 0.0063, p = 0.9370$). Thus, it is imperative that the entire UI stack of access-granting features be authentic, as is the case with our ACGs.

Multi-Clipboard. In another task, we explored the participants’ use of and preference for a multi-clipboard. In a

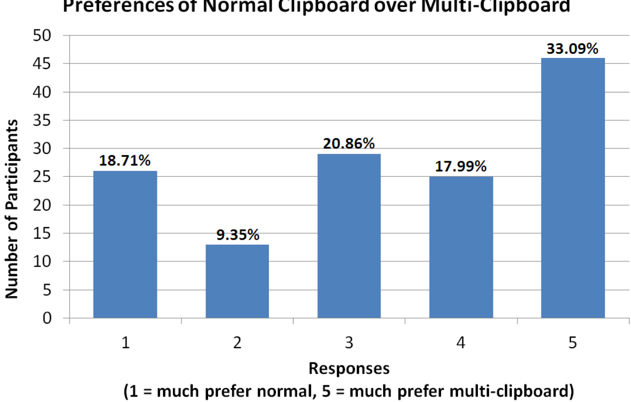

Figure 11: **Responses to a Likert scale question about preference of the multi-clipboard to a normal clipboard.** *These results are statistically significant compared to an even distribution of responses ($\chi^2(N = 139) = 20.2446, p = 0.0004$).*

multi-clipboard, users can copy multiple items in a row without pasting in between, and then find all of them in the clipboard's multi-object history. This allows users to, for instance, copy multiple items in one application and then paste them one by one into another application, needing to switch windows between the two only once. Prior work [37] has suggested such a clipboard based on users' copy-and-paste habits, and Microsoft Office supports a multi-object clipboard, as do various custom clipboard extensions (e.g., [8]). The goal of this task was to observe how (or if) participants responded to attack scenarios (errors in copied text or additional copied text appearing).

In general, we found that people liked the multi-clipboard. Figure 11 shows a distribution of participants' preference indication of the multi-clipboard over a normal clipboard. We also coded participant responses to a free-form question in the feedback for this task and found that 22% of participants explicitly indicated that the multi-clipboard would be useful. For example, one participant said: *I use copy and paste a lot in my work, so this function is fantastic.*

The multi-clipboard appeared to aid users in avoiding error or attacks. In the attack scenarios, over 56% of participants still succeeded at the task (by contrast, over 96% succeeded at non-attack trials). This means that they noticed the false or additional word and either did not paste it or manually edited it after pasting. Thus, UI/X techniques like the multi-clipboard and the drag visuals that we describe in the next paragraphs may supplement user-driven access control to aid users in forming correct mental models and avoiding attacks.

Drag Visuals. In the third task, we studied the effectiveness of visual cues during a drag-and-drop activity on user awareness of attack scenarios. In current web browsers, little information is provided when a user drags an item. The cursor changes to a generic "cannot drop here" cursor until the mouse is over an area in which the item can be dropped, at which point it changes to a generic "can drop here" cursor. The lack of information about which item is being dragged – and from what source – enables an opportunity for data theft. We devised a task in which the participant was asked to drag ten balls and drop them into a seal's nose; in one trial (randomized per participant), the drag icon showed a Gmail envelope icon instead of a ball.

A large majority (71%) of participants reported noticing the inconsistency. However, most did not change their behavior, dropping the object on the seal's nose anyway. Only 14.19% of all participants succeeded under attack conditions. Of the participants that reported noticing inconsistences, 13.76% succeeded; (reportedly) noticing the inconsistency did not have a significant effect on success ($\chi^2(N = 137) = 0.241, p = 0.6235$).

These results indicate that the drag visual is a promising technique to help users avoid error and attack, as they do notice inconsistencies. However, users would need to be trained on the meaning of the visual, as most users who noticed the inconsistency disregarded it.

7 Related Work

Philosophically, our user-driven access control aligns with Yee's proposals to align usability and security in the context of capability systems [45]. Below, we compare our techniques with other systems.

7.1 User-Owned Resource Access Control

Desktops. Desktop operating systems provide many options for accessing user-owned resources and data. Examples include a global file system, cross-application objects sharing via OLE and COM in Windows, or the global clipboard in the X Window System. These mechanisms support many application-agnostic access control abstractions but are terribly insecure: all user-owned resources are accessible to any application at almost any time. By contrast, we provide strong security properties with nearly the same flexibility as desktop systems.

Commercial Browsers. Commercial browsers, including IE, Firefox, and Chrome, isolate the user's local machine from websites, e.g., by restricting access to the file system and to the clipboard [19]. While this model helps prevent attacks on user-owned resources, it restricts website functionality. Recent specifications for access to user-owned resources [39, 41] note only that permission-granting should be tied to explicit user actions but do not address how these should be mapped to system-level access control decisions.

Browser Plugins. Browser restrictions have led web developers to use browser plugins and extensions to access user-owned resources, thereby violating least privilege. For instance, web developers have created copy and paste buttons by overlaying transparent Flash elements and essentially clickjacking [26]. A recent version of Flash introduced a user-initiated action requirement [1], restricting paste to the Ctrl-V shortcut and requiring a click or keystroke for other permissions. Silverlight has a similar model [20]. These requirements do not accurately capture user intent, as users often perform clicks or keystrokes unrelated to the access granted.

Smartphones. As discussed in Section 2, smartphone OSes like iOS [3] and Android [2] allow users to grant access via prompts or manifests.

Experimental Browsers and Browser OSes. Research systems, including experimental browsers [7, 13, 42] and browser operating systems [38, 43], focus on strongly isolating web applications but do not address the problem of intuitive, user-driven access control. The authors of Tahoma point out the need for user-driven access control, but do not further explore how users grant access nor the semantics of such accesses.

7.2 Trusted Window Systems

EROS/EWS and Qubes. The EROS Trusted Window System (EWS) [36] and Qubes OS [33] examine issues similar to those considered in this work. Like Flash, both EWS and Qubes use a specific gesture to indicate copy and paste. EWS also supports drag-and-drop. We generalize these notions into generic support for kernel-recognized gestures. Additionally, 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.

NitPicker. NitPicker [12] is another trusted window system; it supports only drag-and-drop. The authors claim that this is because copy-and-paste is a problem independent of a trusted GUI and could be supported with a trusted clipboard component with which applications can communicate. Of course, this does not address how a user expresses their intent to copy and paste, nor does it consider access to other user-owned resources.

7.3 Multi-Level Security and Information Flow Control

Multi-level security systems like SELinux [34] classify information and users into sensitivity levels to support mandatory access control security policies. Information flow control techniques [17, 46] allow applications or users to express information flow policies explicitly. These policies are enforced by the operating system or language runtime.

Unlike these approaches, user-driven access control does not require the specification of explicit policies; access control decisions are inferred from user actions during natural interaction with the system and applications.

8 Conclusion

In this paper we introduced *user-driven access control*, whereby the operating system infers a user's intent to grant an application access to his or her resources. This intent is inferred from the user's natural interaction with the

system and applications. To support user-driven access control in modern operating systems, we introduced two OS techniques, access control gadgets and kernel-recognized gestures. These techniques are general and readily support user-driven access control of all user-owned resources, including privacy- and cost-sensitive devices, settings, and content. Systems can thereby support a broad range of access-control metaphors, including drag-and-drop, copy-and-paste, content picking, and global search. Our implementation and evaluation indicate that by accurately capturing user intent, user-driven access control adds the functionality of the traditional desktop to the security of isolated application systems.

9 Acknowledgements

I sincerely thank my collaborators for all of their help and hard work on this project. I also thank Crispin Cowan, Stuart Schechter, Steve Gribble, and the UW CSE systems seminar for their valuable feedback on this work. Thanks to Greg Akselrod, Roxana Geambasu, and Dan Halperin for their feedback on earlier drafts. I thank Michael Berg and Microsoft User Research, Andrew Begel, James Fogarty, Batya Friedman, and my CSE 510 (HCI) classmates for their help with the user study; thanks to the study participants for their participation. Finally, thanks to Cornel Lupu and Vince Orgovan for their help with obtaining Windows application popularity data.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124. This work was done in part while employed by Collabera as a contractor for Microsoft Research.

References

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

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

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

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

[5] 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).

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

[7] 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).

[8] DITTO CLIPBOARD MANAGER. http://ditto-cp.sourceforge.net/.

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

[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 *To Appear in USENIX Conference on Web Application Development* (June 2011).

[12] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *21st Annual Computer Security Applications Conference* (2005).

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

[14] HARRIS, J. No Distaste for Paste (Why the UI, Part 7), April 2006. http://blogs.msdn.com/b/jensenh/archive/2006/04/07/570798.aspx.

[15] 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).

[16] 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), 1147–1165.

[17] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *21st Symposium of Operating Systems Principles* (2007).

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

[19] MICROSOFT. How to Prevent Web Sites From Obtaining Access to the Contents of Your Windows Clipboard, March 2007. http://support.microsoft.com/kb/224993.

[20] MICROSOFT. Silverlight Clipboard Class, 2010. http://msdn.microsoft.com/en-us/library/system.windows.clipboard%28v=VS.95%29.aspx.

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

[22] MICROSOFT USER RESEARCH. http://www.microsoft.com/usability/.

[23] MOSHCHUK, A., WANG, H. J., AND LIU, Y. Content-Based Principal Model: Rethinking Isolation in Modern Client Systems. In *Submission to the 18th ACM Symposium on Operating Systems Principles* (2011).

[24] MOTIEE, S., HAWKEY, K., AND BEZNOSOV, K. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In *6th Symposium on Usable Privacy and Security* (2010).

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

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

[27] OAUTH. http://oauth.net.

[28] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[29] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *ACM ASPLOS Conference* (2011).

[30] REEDER, R., KELLEY, P., MCDONALD, A., AND CRANOR, L. A User Study of the Expandable Grid Applied to P3P Privacy Policy Visualization. In *ACM Workshop on Privacy in the Electronic Society (WPES)* (2008).

[31] REIS, C., AND GRIBBLE, S. D. Isolating Web Programs in Modern Browser Architectures. In *ACM EuroSys Conference* (2009).

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

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

[34] SERVICE, N. C. S. Security-Enhanced Linux, January 2009. http://www.nsa.gov/research/selinux/index.shtml.

[35] 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 Conference* (2007).

[36] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS Trusted Window System. In *13th USENIX Security Symposium* (2004).

[37] STOLEE, K. T., ELBAUM, S., AND ROTHERMEL, G. Revealing the Copy and Paste Habits of End Users. *IEEE Visual Languages and Human-Centric Computing* (2009).

[38] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *9th Symposium on Operating Systems Design and Implementation* (2010).

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

[40] W3C. HTML5: File API, 2011. http://www.w3.org/TR/FileAPI/.

[41] W3C. Web Applications 1.0: The Device Element, 2011. http://dev.w3.org/html5/html-device.

[42] 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 *18th USENIX Security Symposium* (2009).

[43] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of Desktop and Web Applications on a Multi-Service OS. In *4th USENIX Hot Topics in Security* (2009).

[44] XIONG, X., TIAN, D., AND LIU, P. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. In *Network and Distributed System Security Symposium* (2011).

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

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