Securing Embedded User Interfaces: Android and Beyond

Franziska Roesner and Tadayoshi Kohno, *University of Washington*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

Securing Embedded User Interfaces: Android and Beyond

Franziska Roesner and Tadayoshi Kohno
University of Washington

Abstract

Web and smartphone applications commonly embed third-party user interfaces like advertisements and social media widgets. However, this capability comes with security implications, both for the embedded interfaces and the host page or application. While browsers have evolved over time to address many of these issues, mobile systems like Android — which do not yet support true cross-application interface embedding — present an opportunity to redesign support for secure embedded user interfaces from scratch. In this paper, we explore the requirements for a system to support secure embedded user interfaces by systematically analyzing existing systems like browsers, smartphones, and research systems. We describe our experience modifying Android to support secure interface embedding and evaluate our implementation using case studies that rely on embedded interfaces, such as advertisement libraries, Facebook social plugins (e.g., the "Like" button), and access control gadgets. We provide concrete techniques and reflect on lessons learned for secure embedded user interfaces.

1 Introduction

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

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

The Web security model has evolved over time to address these and other threats. For example, the same-origin policy prevents embedded content from directly accessing or manipulating the parent page, and vice versa. As recently as 2010, browsers have added the `sandbox` attribute for iframes [1], allowing websites to prevent embedded content from running scripts or redirecting the top-level page. However, other attacks — like clickjacking — remain a serious concern. Malicious websites frequently mount "likejacking" attacks [24] on the Facebook "Like" button, in which they trick users into sharing the host page on their Facebook profiles. If Facebook suspects a button of being part of such an attack, it asks the user to confirm any action in an additional popup dialog [7] — in other words, Facebook falls back on a non-embedded interface to compensate for the insecurity of embedded interfaces.

While numerous research efforts have attempted to close the remaining security gaps with respect to interface embedding on the Web [11, 25, 29], they struggle with maintaining backwards compatibility and are burdened with the complexity of the existing Web model. We argue that Android, which to date offers no cross-application embedding, offers a compelling opportunity to redesign secure embedded interfaces from scratch.

Today, applications on Android and other mobile operating systems cannot embed interfaces from another principal; rather, they include third-party libraries that run in the host application's context and provide custom user interface elements (such as advertisements). On the one hand, these libraries can thus abuse the permissions of or otherwise take advantage of their host applications. On the other hand, interface elements provided by these libraries are vulnerable to manipulation by the host application. For example, Android applications can programmatically click on embedded ads in an attempt to increase their advertising revenue [18]. This lack of security also precludes desirable functionality from the Android ecosystem. For example, the social plugins that Facebook provides on the Web (e.g., the "Like" button or comments widget) are not available on Android.

Previous research efforts for Android [17, 23] have focused only on one interface embedding scenario: advertising. As a result, these systems, while valuable, do not provide complete or generalizable solutions for interface embedding. For example, to our knowledge, no existing Android-based solution prevents a host application from eavesdropping on input to an embedded interface.

In this paper, we explore what it takes to support secure embedded UIs on Android. We systematically analyze existing systems, including browsers, with respect

to whether and how they provide a set of security properties. We view this analysis and the framework we use for it as a contribution in its own right. Informed by this analysis, we describe our experiences modifying the Android framework to support cross-principal interface embedding in a way that meets our security goals. We evaluate our implementation using case studies that rely on embedded interfaces, including: (1) advertisement libraries that run in a separate process from the embedding application, (2) Facebook social plugins, to date available only on the Web, and (3) access control gadgets [19] that allow applications to access sensitive resources (like geolocation) only in response to real user input.

Through our implementation experience, we consolidate and evaluate approaches from prior work. We find that some techniques can be simplified in practice — such as an approach for maintaining invariants in the UI layout tree [18] — but that we face additional practical challenges, like propagating layout changes across processes. We discover that an embedded element's size is an important factor in preventing clickjacking, as well as that we can apply prior work on access control gadgets [19] in novel ways to improve interaction flexibility beyond the browser model. We discuss these and other challenges and lessons in more detail in Section 8, which benefits from the context of the preceding sections.

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

2 Motivation and Background

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

2.1 Functionality

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

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

System UI. Security-sensitive actions often elicit system interfaces, usually in the form of prompts. For example, Windows users are shown a User Account Control dialog [14] when an application requires elevation to administrative privileges, and iOS and browser users must respond to a permission dialog when an application attempts to access a sensitive resource (e.g., geolocation).

Because prompts result in a disruptive user experience, the research community has explored using embedded system interfaces to improve the usability of security-sensitive interactions like resource access. In particular, a recent paper [19] describes access control gadgets (ACGs), embeddable UI elements that — with user interaction — grant applications access to various system resources, including the camera, the clipboard, the GPS, etc. For example, an application might embed a location ACG, which is provided by the system and displays a recognizable location icon; when the user clicks the ACG, the embedding application receives the current GPS coordinates. As we describe below, ACGs cannot be introduced into most of today's systems without significant changes to those systems.

2.2 Threat Model and Security Concerns

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

We observe that while Web and smartphone applications rely heavily on third-party content and services, the associated third-party user interface is not always actu-

ally embedded inside of the client application. For example, websites redirect users to PayPal's full-screen page, OAuth authorization dialogs appear in pop-up or redirect windows, and Web users who click on a Facebook "Like" button that is suspected of being part of a clickjacking attack will see an additional pop-up confirmation dialog. We observe two main security-related reasons for the choice not to embed or not to be embedded.

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

Our goal in this paper is *not* to address such phishing attacks, but rather to evaluate and implement methods for securely embedding one legitimate (i.e., not spoofed) application within another. (While extensions of existing approaches, such as SiteKeys, may help mitigate embedded phishing attacks, these approaches do have limitations [21] and are orthogonal to the goals of this paper.[1])

More importantly — and the subject of this paper — even legitimate embedded interfaces may be subject to a wide range of attacks, or may present a threat to the application or page that embeds them. In particular, drawing in part on [18], embedded interfaces or their parents may be subject to:

Display forgery attacks, in which the parent application modifies the child element (e.g., to display a false payment value), or vice versa.

Size manipulation attacks, in which the parent application violates the child element's size requirements or expectations (e.g., to secretly take photos by hiding the camera preview [26]), or the child element sets it own size inappropriately (e.g., to display a full-screen ad).

Input forgery attacks, in which the parent application delivers forged user input to a child element (e.g., to programmatically click on an advertisement to increase ad revenue), or vice versa.

Clickjacking attacks, in which the parent application forces or tricks the user into clicking on an embedded element [11] using visual tricks (partially obscuring the child element or making it transparent) or via timing-based attacks (popping up the child element just as the user is about to click in a predictable place).

Focus stealing attacks, in which the parent application steals the input focus from an embedded element, capturing input intended for it, or vice versa.

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

Ancestor redirection attacks, in which a child element redirects an ancestor (e.g., the top-level) application or page to a target of its choice, without user consent.

Denial-of-service attacks, in which the parent application prevents user input from reaching a child element (e.g., to prevent a user from clicking "Cancel" on an authorization dialog), or vice versa.

Data privacy attacks, in which the parent or child extract content displayed in the other.

Eavesdropping attacks, in which the parent application eavesdrops on user input intended for a child element (e.g., a sensitive search query), or vice versa.

2.3 Security Goals

Motivated by the above challenges and building on recent work [18], we now describe the security goals that we apply in our analysis and implementation. Where noted, we describe additional goals not discussed by prior work.

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

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

3 The Case for Secure UIs in Android

While Section 2 considered UI embedding in general, we now specifically make the case for secure embedded UIs in Android. The fact that an Android application cannot embed another application's interface results in a

fundamental trust assumption built into the Android UI toolkit. In particular, every UI element trusts its parent and its children, who each have unrestricted access to the element's APIs. Vulnerabilities arise when this trust assumption is violated, e.g., because an embedded element is provided by a third-party library.

We now introduce several case studies illustrating that embedded user interface scenarios in stock Android are often either insecure or impossible. We will return to these case studies in Section 6 and reevaluate them in the context of our implementation.

Advertising. In stock Android, applications wishing to embed third-party advertisements must include an ad library, such as AdMob or Mobclix, which runs in the embedding application's process. These libraries provide a custom UI element (an AdView) that the embedding application instantiates and embeds. As has been discussed extensively in prior work [17, 23], the library model for third-party advertisements comes with a number of security and privacy concerns. For example, the host application must trust the advertising library not to abuse the host's permissions or otherwise exploit a buggy host application. Additionally, ad libraries ask their host applications to request permissions (such as location and Internet access) on their behalf; applications that request permissions not clearly relevant to their stated purpose can desensitize users to permission warnings [8].

Prior work [18] has also identified and experimentally demonstrated threats to the AdView. Parent applications can mount a programmatic clickfraud attack in which they programmatically click on embedded ads to increase their advertising revenue. Similarly, parent applications can mount clickjacking attacks by, for example, covering the AdView with another UI element that does not accept (and thus lets pass through) input events.

WebViews. One of the built-in UI elements provided by Android is the WebView, which can load local HTML content or an arbitrary URL from the Internet. Though WebViews appear conceptually similar to iframes, they do not provide many of the same security properties. In particular, WebViews — and more importantly, the contained webpage — can be completely manipulated by the containing application, which can mount attacks including programmatic clicking, clickjacking, and input eavesdropping [13]. Thus, for example, if an Android application embeds a WebView that loads a login page, that application can eavesdrop on the user's password as he or she enters it into the WebView.

Facebook Social Plugins. On the Web, Facebook provides a set of social plugins [6] to third-party web developers. These plugins include the "Like" button, a comments widget, and a feed of friends' activities on the embedding page (e.g., which articles they liked or shared). These social plugins are generally implemented as iframes and thus isolated from the embedding page.

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

Access Control Gadgets. Finally, recent work [19] has proposed access control gadgets (ACGs), secure embedded UI elements that are used to capture a user's permission-granting intent (e.g., to grant an application access to the user's current location). Authentically capturing a user's intent relies on a set of UI-level security properties including clickjacking protection, display isolation, and user intent protection. As we describe in this paper, fundamental modifications to Android are required to enable secure embedded elements like ACGs.

4 Analysis of UI Embedding Systems

To assess the spectrum of solutions and to inform our implementation choices, we now step back and analyze prior Web and Android based solutions for cross-application embedded interfaces with respect to the set of security properties described in Section 2.3. This analysis is summarized in Figure 1.

4.1 Browsers

Browsers support third-party embedding by allowing web pages to include iframes from different domains. Like all pages, iframes are isolated from their parent pages based on the same-origin policy [27], and browsers do not allow pages from different origins to capture or generate input for each other.

However, an iframe's parent has full control of its size, layout, and style, including the ability to make it transparent or overlay it with other content. These capabilities enable clickjacking attacks. While there are various "framebusting" techniques that allow a sensitive page to prevent itself from being framed in an attempt to prevent such attacks, these techniques are not foolproof [20]. More importantly, framebusting is a technique to prevent embedding, not one that supports secure embedding.

Additionally, while an iframe cannot read the URL(s) of its ancestor(s), it can change the top-level URL, redirecting the page without user consent. Newer version of

Category	*Security Requirement*	**Browsers**	**Android**	**AdDroid [17]**	**AdSplit [23]**	**RFK [18]**
Display Integrity	Prevents direct modification	✓	✗	✗	✓	✓*
	Prevents size manipulation	✗	✗	✗	✗	✓*
Input Integrity	Prevents programmatic input	✓	✗	✗	✓	✓*
Intent Integrity	Clickjacking protection	✗	✓	✓	✓	✓
	Prevents input denial-of-service	✓	✗	✗	✗	✓*
	Prevents focus stealing	✓	✗	✗	✓	✗
	Prevents ancestor redirection	✓	✗	✗	✗	✗
Data Isolation	Prevents access to display	✓	✗	✗	✓	✓*
	Prevents input eavesdropping	✓	✗	✗	✗	✓*
UI-to-API Links	APIs can verify caller	✓	✗	✗	✓	✓*

Figure 1: **Analysis of Existing Systems.** *This table summarizes, to the best of our knowledge, the UI-level security properties (first defined in prior work [18] and expanded here) achieved by existing systems. Figure 2 similarly analyzes our implementation. * Checkmarks annotated with an asterisk require static analysis or hypothetical (not prototyped) changes to the Android framework.*

some browsers allow parent pages to protect themselves by using the `sandbox` attribute for iframes; thus, we've indicated that the Web prevents such attacks in Figure 1. However, we observe that it may be desirable to allow user actions to override such a restriction, and we describe how to achieve such a policy in later sections.

Research browsers and browser operating systems (e.g., Gazelle [29] and IBOS [25]) provide similar embedded UI security properties as traditional browsers, and thus we omit them from Figure 1. Gazelle partially addresses clickjacking by allowing only opaque cross-origin overlays, but this policy is not backwards compatible. Furthermore, malicious parent pages can still obscure embedded content by partially overlaying additional content on top of sensitive iframes. We discuss additional work considering clickjacking in Section 9.

4.2 Android

Two recent research efforts [17, 23] propose privilege separation to address security concerns with Android's advertising model (under which third-party ad libraries run in the context of the host application). AdDroid's approach [17] introduces a system advertising service that returns advertisements to the AdDroid userspace library, which displays them in a new user interface element called an AdView. While this approach successfully removes untrusted ad libraries from applications, it does not provide any additional UI-level security properties for the embedded AdView beyond what is provided by stock Android (see Figure 1). For example, it does not prevent the host application from engaging in clickfraud by programmatically clicking on ads.

AdSplit [23], on the other hand, fully separates advertisements into distinct Android applications (one for each host application). AdSplit achieves the visual embedding of the ad's UI into the application's UI by overlaying the host application, with a transparent region for the ad, on top of the ad application. It prevents programmatic clickfraud attacks by authenticating user input using Quire [3]. As summarized in Figure 1, AdSplit meets the majority of security requirements for embedded UIs. Indeed, the requirements it meets are sufficient for embedded advertisements. Because it does not meet all of the requirements, however — most importantly, it does not prevent input eavesdropping — AdSplit would not be well-suited as a generalized solution for embedded UIs.

Finally, the prototype implementation described in [18] to meet that work's goals (upon which we build) also contains weaknesses. In particular, the isolation and identification of different principals ("trust groups" in the terminology of that paper) is insecure, undermining all of the security properties. Rather than truly supporting one Android application embedding UI from another application, it merely separates interfaces defined in the main application from those defined in included libraries. This separation relies on Java package names, static code analysis, and hypothetical changes to the Android framework (e.g., changing Android's Java classloader to enable package sealing) that have not been implemented or verified in practice.

5 Implementation Experience: LayerCake

We now explore what it takes to support secure embedded UIs, under the definitions from Section 2.3, in the Android framework. As no existing Android-based solutions meet these goals, we view this implementation as an opportunity to consider secure embedding from scratch. While we adapt techniques from prior work, we find that previously published guidelines are not always directly applicable. For example, we found that we could simplify a prior approach [18] when overlaying cross-application content, but that we faced additional practical challenges, such as the need to propagate layout changes

Category	*Security Requirement*	**LayerCake**	*(Section Number) Approach*
Display Integrity	Prevents direct modification	✓	*(5.3)* Embedded elements in isolated, overlaid windows.
	Prevents size manipulation	✓	*(5.6)* User notifications on size conflicts.
Input Integrity	Prevents programmatic input	✓	*(5.3)* Embedded elements in isolated, overlaid windows.
Intent Integrity	Clickjacking protection	✓	*(5.7)* No input delivered if view/window not fully visible.
	Prevents input denial-of-service	✓	*(5.3)* Embedded windows attached to system root.
Data Isolation	Prevents access to display	✓	*(5.3)* Embedded elements in isolated, overlaid windows.
	Prevents input eavesdropping	✓	*(5.3)* Embedded windows attached to system root.
	Prevents focus stealing	✓	*(5.4)* Focus changes only in response to real user clicks.
	Prevents ancestor redirection	✓	*(5.8)* Prompts and *(6.2)* redirection ACG.
UI-to-API Links	APIs can verify caller	✓	*(5.2)* Elements from different principals run in separate calling processes (identifiable by package name).

Figure 2: **Techniques for Secure Embedded UI.** *This table summarizes how LayerCake (our modified version of Android 4.2) achieves each of the desired security properties for embedded user interface elements.*

and handle multiple levels of nesting. We further discuss these and other challenges and lessons in Section 8.

We thus created *LayerCake*, a modified version of the Android framework that supports cross-application embedding via changes to the ActivityManager, the WindowManager, and input dispatching. We added or modified 2400 lines of source code across 50 files in Android 4.2 (Jelly Bean). Figure 2 summarizes the implementation choices that achieve our desired security properties.

5.1 Android Background

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

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

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

While only one Activity can be in the foreground, multiple applications/processes may have visible windows.

Figure 3: **Sample Application.** *This restaurant review application embeds two third-party Activities, an advertisement and a map. The map Activity further embeds an access control gadget (ACG) for location access.*

For example, the status bar runs in the system process, and the window of one application may be visible below the (partially) transparent window of another. As an example of the latter, AdSplit [23] achieves visual embedding by taking advantage of an application's ability to make portions of its UI transparent. However, recall from Section 4 and Figure 1 that this approach is insufficient for generalized embedded UI security.

5.2 Supporting Embedded Activities

LayerCake introduces a new View into Android's user interface toolkit (Java package `android.view`) called `EmbeddedActivityView`. It allows an application developer to embed another application's Activity within her application's interface by specifying in the parameters of the EmbeddedActivityView the package and class

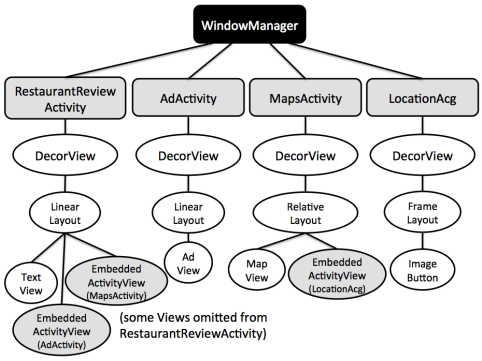

Figure 4: **Window Management.** *This figure shows the Window/View tree for the Activities in Figure 3. Embedded Activities are not embedded in the View tree (circles) of their parent, but rather within a separate window. The WindowManager keeps track of a window (grey squares) for each Activity and visually overlays an embedded window on top of the corresponding EmbeddedActivityView in the parent Activity.*

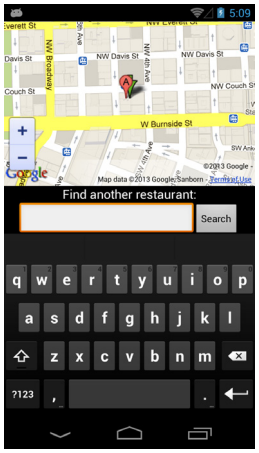

Figure 5: **Panning for Software Keyboard.** *The restaurant review application (from Figure 3), including its overlaid embedded windows, must be panned upward to make room for the software keyboard underneath the in-focus text box.*

names of the desired embedded Activity. Figure 3 shows a sample application that embeds several Activities.

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

An Activity may embed multiple other Activities, which themselves may embed one or more Activities (multiple nesting). Each embedded Activity is started as a new instance, so multiple copies of the same Activity are independent (although they run in the same application, allowing changes to the application's global state to persist across different Activity instances).

5.3 Managing Windows

Properly displaying embedded Activities required modifications to the Android WindowManager (Java). One option for achieving embedded UI layouts is to literally nest them — that is, to add the embedded Activity's Views (UI elements) as children in the parent Activity's UI tree. However, this design would allow the parent Activity to mount input eavesdropping and denial-of-service attacks on the child Activity. Thus, following the interface layout tree invariants described in prior work [18], we do not literally nest the interface elements of embedded Activities inside the parent Activity. Instead, an embedded Activity is displayed in a new window, overlaid on top of the window to which it is attached (i.e., the window of the parent Activity). This overlay achieves the same visual effect as literal embedding but prevents input manipulation attacks. Figure 4 shows an example of the interface layout trees associated with the Activities in the sample application in Figure 3. We note that we were able to simplify the proposed approach [18], which we found to be overly general (see Section 8).

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

The relative position and size of an overlaid window are specified by the embedding application in the layout parameters of the EmbeddedActivityView and are honored by the WindowManager. (Note that the specified size may violate size bounds requested by the embedded Activity, as we discuss in Section 5.6.)

The layout parameters of an embedded Activity's window must remain consistent with those of the associated EmbeddedActivityView, a practical challenge not described in prior work. For example, when the user reorients the phone into landscape mode, the parent Activity will adjust its UI. Similarly, when the soft keyboard is shown, Android may pan the Activity's UI upwards in order to avoid covering the in-focus text box with the keyboard (Figure 5). In both cases, the embedded Activity's windows must be relocated appropriately. To support these dynamic layout changes, the EmbeddedActivityView reports its layout changes to the WindowManager, which applies them to the associated window.

Finally, since LayerCake supports multiple levels of embedding, it must appropriately display windows multiple levels down (e.g., grandchildren of the top-level Ac-

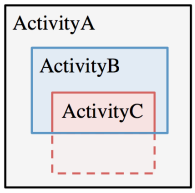

Figure 6: **Cropping Further Nested Activities.** *If a grandchild (ActivityC) of the top-level Activity (ActivityA) is placed or scrolled partly out of the visible area of its immediate parent (ActivityB), it must be cropped accordingly.*

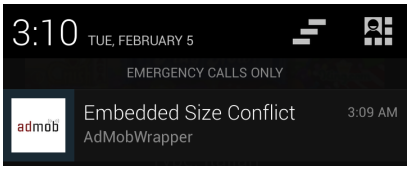

Figure 7: **Size Conflict Notification.** *If the AdMobWrapper application specifies a minimum size that the RestaurantReviewActivity does not honor when it embeds the advertisement, a system notification is displayed to the user. Clicking on the notification displays a full-screen advertisement Activity.*

tivity). For example, suppose ActivityA embeds ActivityB which embeds ActivityC. If the EmbeddedActivityView (inside ActivityB) that corresponds to ActivityC is not fully visible — e.g., because it is scrolled halfway out of ActivityB's visible area — then the window corresponding to ActivityC must be cropped accordingly (Figure 6). This cropping is necessary because ActivityC is not literally nested within ActivityB, but rather overlaid on top of it, as discussed above.

5.4 Handling Focus

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

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

5.5 Supporting Cross-Principal APIs

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

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

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

5.6 Handling Size Conflicts

Recall from Section 5.3 that the WindowManager honors the parent application's size specification for an EmbeddedActivityView. This policy prevents a child element from taking over the display (a threat discussed further in the context of ancestor redirection below). However, we also wish to prevent size manipulation by the parent.

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

Prior work [18] describes different size conflict policies based on whether the embedded element is trusted or untrusted by the system. If it is trusted (e.g., a system-defined ACG), its own size request should be honored; if it is untrusted (e.g., an ad that requests a size filling

the entire screen), the parent's size specification is honored. However, we observe that a malicious parent can mimic the effect of making a child element too small using other techniques, such as scrolling it almost entirely off-screen — and that doing so maliciously is indistinguishable from legitimate possible scroll placements. We thus further consider the failure to meet minimum size requirements in the context of clickjacking (Section 5.7).

Thus, since enforcing a minimum size for trusted embedded elements does provide additional security properties in practice, we use the same policy no matter whether mis-sized elements are trusted or untrusted by the system. That is, the WindowManager honors the size specifications of the parent Activity. If these values are smaller than the embedded Activity's request, a status bar notification is shown to the user (Figure 7). Similar to a browser's popup blocker, the user can click this notification to open a full-screen (non-embedded) version of the Activity whose minimum size was not met.

5.7 Support for Clickjacking Prevention

In a clickjacking attack [11], a malicious application forces or tricks a user into interacting with an interface, generally by hiding important contextual information from the user. For instance, a malicious application might make a sensitive UI element transparent or very small, obscure it with another element that allows input to pass through it, or scroll important context off-screen (e.g., the preview associated with a camera button).

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

1. *Covered (fully or partly) by another window.* This request is already supported by stock Android via `setFilterTouchesWhenObscured()`.
2. *Not the minimum requested size.* A parent application may not honor a child's size request (see Section 5.6).
3. *Not fully visible due to window placement.* An embedded Activity's current effective window may be cropped due to scrolling.

Note that an embedded Activity need not be concerned about a malicious parent making it transparent, because stock Android already does not deliver input to invisible windows. Similarly, an Activity need not be concerned about malicious visibility changes to UI elements within its own window, since process separation ensures that the parent cannot manipulate these elements. To prevent timing-based attacks, these criteria should be met for some minimum duration [11] before input is delivered, a check that we leave to future work.

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

5.8 Preventing Ancestor Redirection

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

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

Second, we introduce a prompt shown to users when an embedded Activity attempts to launch another Activity full-screen (i.e., not using the flag described above). This may happen either because it is a legacy application unaware of the flag, or for legitimate reasons (e.g., a user's click on an embedded advertisement opens a new browser window). However, studies have shown that prompting users is disruptive and ineffective [16]; in Section 6.2 we discuss an access control gadget (ACG) that allows embedded applications to launch full-screen Intents in response to user clicks without requiring that the system prompt the user.

6 Case Studies

We now return to the case studies introduced in Section 3 and describe how LayerCake supports these and other scenarios. Figure 8 shows that implementation complexity is low, especially for parent applications.

6.1 Geolocation ACG

To support user-driven access for geolocation, we implemented a geolocation access control gadget (ACG) in the spirit of prior work [19]. We added a `LocationAcg` Activity to Android's SystemUI (which runs in the system process and provides the status bar, the recent applications list, and more). This Activity, which other applications can embed, simply displays a location button (see Figure 3).

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

Figure 8: **Implementation Complexity.** *Lines of code for (1) the embedded Activity and (2) the parent's implementation of the AIDL interface. We omit legacy applications because they required no modifications and expose no parent interfaces. Implementation complexity is low, especially for embedders.*

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

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

We emphasize again that this ACG provides location information to the parent application only when the user wishes to share that information; a well-behaving parent application will not need location permissions. In a system like Android, where applications can request location permissions in their manifest, it is an open question how to incentivize developers to use the corresponding ACG instead of requesting that permission. Prior work [19] has suggested incentives including increased scrutiny at app store review time of applications requesting sensitive permissions.

6.2 Redirection Intent ACG

In Section 5.8, we introduced a system prompt when an embedded Activity attempts to start a full-screen Activity. However, prompts are known to be disruptive and often ignored, especially following a user action intended to cause the effect about which the prompt warns [31]. For example, a user who clicks on an embedded ad in stock Android today expects it to open the ad's target in a new (non-embedded) browser window. Following the philosophy of user-driven access control [19], we thus allow embedded Activities to start top-level Activities without a prompt if `startActivity()` is called in response to a user's click.

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

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

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

Note that developers are incentivized to use the RedirectAcg because otherwise attempts to launch top-level Activities will result in a disruptive prompt (Section 5.8).

6.3 Secure WebView

We implemented a SecureWebView that addresses security concerns surrounding Android WebViews [12, 13]. The SecureWebView is an Activity in a new built-in application (WebViewApp) that consists solely of an ordinary WebView (inside a FrameLayout) that fills the Activity's whole UI. Thus, when another Activity embeds a SecureWebView, the internal WebView takes on the dimensions of the associated EmbeddedActivityView.

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

Security Discussion. Separating out the Android WebView into another process — that of the WebViewApp — provides important missing security properties. It is no longer possible to eavesdrop on input to the embedded

webpage, to extract content or programmatically issue input, or to manipulate the size, location, or transparency of the WebView to mount clickjacking attacks.

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

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

6.4 Advertisements

Recall that stock Android applications embedding third-party advertisements include an ad library that runs in the host application's process and provides an AdView element. Our modifications separate the AdView out into its own process (see the advertisement in Figure 3). To do this, we create a wrapper application for the AdMob advertising library [10]. The wrapper application exposes an embeddable Activity (called `EmbeddedAd`) that instantiates an AdMob AdView with the specified parameters. This Activity exposes all of AdMob's own APIs across the process boundary, allowing the embedding application to specify parameters for the ad.

Security Discussion. Moving ads into their own process (one process per ad library) addresses a number of the concerns raised in Section 3. In particular, an ad library can no longer abuse a parent application's permissions or exploit a buggy parent application. Furthermore, the permissions needed by an ad library, such as Internet and location permissions, must no longer be requested by the parent application (unless it needs these permissions for other purposes).

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

LayerCake goes beyond process separation, providing UI-level security absent in most prior systems (except AdSplit [23]). Most importantly, the parent can no longer mount programmatic click fraud attacks.

6.5 Facebook Social Plugins

We can now support embedded Facebook social widgets in a secure manner. We achieve this by creating a Facebook wrapper application that exposes Activities for various Facebook social widgets (e.g., a Comments Activity and a Like Activity — see Figure 9). Each Activity displays a WebView populated with locally-generated HTML that references the Facebook JavaScript SDK to generate the appropriate plugin (as done ordinarily by web pages and as specified by Facebook [6]).

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

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

6.6 Legacy Applications

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

To demonstrate this, we created an application that embeds both the existing Pandora application and the existing Amazon application. To do so, we needed to discover the names of the corresponding Activities in the existing applications. This information is easy to discover from Android's standard log, which prints information about Intent targets when they are launched. Figure 10

Figure 9: **Facebook Social Plugins.** *This example blog application embeds both a Facebook "Like" button and a comments feed, both running in our FacebookWrapper application.*

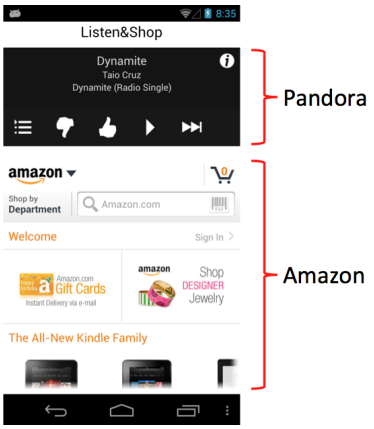

Figure 10: **Embedded Pandora and Amazon Apps.** *Legacy applications can also be embedded, raising policy questions regarding top-level intents and embedding permissions.*

shows a screenshot of the resulting application.

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

Legacy applications naturally do not use the new `FLAG_ACTIVITY_EMBEDDED` flag when launching internal Activities. While updated versions of Pandora and Amazon could use this flag to redirect within an embedded window, the experience with unmodified legacy applications is likely to be disruptive. Thus, a possible policy (perhaps subject to a user preference setting) for such applications is to internally modify all Activity launches to use the new flag, never allowing these applications to break out of their embedded windows.

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

7 Performance Evaluation

We evaluate the performance impact of our changes to Android by measuring the time it takes to start an application, i.e., the delay between a `startActivity()` system call and the `onCreate()` call for the last embedded Activity (or the parent Activity, if none are embedded). As shown in the top of Figure 11, applications with embedded Activities take longer to fully start. The reason for this is that the parent Activity's layout must be created (in its `onCreate()`) before child Activities can be identified. Thus, an application with multiple nested Activities (e.g., RestaurantReviewer) requires linearly more time than an application with only one level of nesting (e.g., FacebookDemo or Listen&Shop). We note that the parent Activity's own load time is unaffected by the presence of embedded content (e.g., the FacebookDemo Activity starts in 160 ms, even though the embedded Facebook components require 300 ms). Prior work [15] has argued that the time to display first content is more important than full load time.

We also measure input event dispatch time (e.g., the time it takes for Android to deliver a touch event to an application). Specifically, we evaluate the impact of dispatching input events first to the WindowManager, allowing it to redirect focus if appropriate (Section 5.4). The bottom of Figure 11 shows that involving the WindowManager in dispatch has a negligible performance impact over stock Android; changing focus has a greater impact, but it is not noticeable by the user, and focus change events are likely rare.

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

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

8 Discussion

Whereas existing systems — particularly browsers — have evolved security measures for embedded user interfaces over time, this paper has taken a principled ap-

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

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

Figure 11: **Performance.** *The top table shows the time it takes for the* `onCreate()` *method of all included Activities to be called. We note that the time to load the parent Activity remains the same whether or not it uses embedding, so the time for the parent to begin displaying native content is unaffected. The bottom table shows that the effect of intercepting input events in the WindowManager for possible focus changes is minor.*

proach to defining a set of necessary security properties and building a system with full-fledged support for embedding interfaces based on these properties.

8.1 Lessons for Embedded Interfaces

From this process, we provide a set of techniques for systems that wish to support secure cross-application UI embedding. Figure 2 outlines the security properties provided by LayerCake and summarizes the implementation techniques used to achieve each property. While prior works [18, 19] have stated the need for many (though not all) of these properties, they have not provided detailed guidelines for implementation. We hope this work, in which we bring techniques from prior work together into a practical implementation, will serve that purpose.

Our implementation experience challenges several previous assumptions or choices. These lessons include:

User-driven ancestor redirection. Embedded applications should not be able to redirect an ancestor application/page without user consent. We argue that a reasonable tradeoff between security and usability is to prompt users only if the redirection attempt does not follow a user click (indicating the user's intent to interact with the embedded content). While newer browsers prevent embedded iframes from redirecting the top-level page programmatically, they do not allow user actions (e.g., clicking on a link with target `_top`) or other mechanisms to override this restriction. In our case studies, we saw that this type of click-enabled redirection can be useful and expected (e.g., when a user clicks on an embedded ad, he or she likely expects to see full-screen content about the advertised product or service). In our system, we were able to apply ACGs in a novel way to capture a user's redirection intent (Section 6.2).

Size manipulation as a subset of clickjacking. We initially considered size manipulation (by the parent of an embedded interface element) to be a stand-alone threat. A solution that we considered is to treat elements that are trusted or untrusted by the system differently (e.g., an access control gadget is trusted while an advertisement is not), letting the system enforce the minimum requested size for trusted elements. However, this solution provides no additional security, since a malicious parent can use other techniques to obscure the sensitive element (e.g., partially covering it or scrolling it partly off-screen). Thus, we consider size manipulation as a subset of clickjacking. We suggest that sufficient size be considered an additional criterion (in addition to traditional clickjacking prevention criteria like complete visibility [11, 19]) for the enabling of a sensitive UI element.

Simplification of secure UI layout tree. Prior work [18] proposes invariants for the interface layout tree that ensure a trusted path to every node and describes how to transform an invalid layout tree into a valid one. Our implementation experience shows this solution to be overly general. Embedded elements need not be attached to the layout tree in arbitrary locations; rather, they can always attach to the (system-controlled) root node and overlaid appropriately by the WindowManager (or equivalent). That is, the layout trees of separate principals need never be interleaved, but rather visually overlaid on top of each other, requiring no complex tree manipulations. Simplifying this approach is likely to make it easier and less error-prone for system developers to support secure embedded UI.

8.2 New Capabilities

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

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

Secure WebViews. It is particularly important that WebViews containing sensitive content run in their own process. While an Android WebView seems at first glance to be similar to an iframe, it does not provide the security properties to which developers are accustomed on the Web (as discussed in this paper and identified in prior

work [12, 13]). LayerCake matches and indeed exceeds the security of iframes — in particular, a SecureWebView can request that the system not deliver user input to it when it is not fully visible or sufficiently large, thereby preventing clickjacking attacks that persist on the Web.

Access Control Gadgets. Prior work [19] introduced ACGs for user-driven access control of sensitive resources like the camera or location, but that work does not provide concrete guidelines for how the necessary UI-level security properties should be implemented. This paper provides these details, and we hope that they will guide system developers to include ACGs in their systems. We particularly recommend that browser vendors consider ACGs in their discussions of how to allow users to grant websites access to sensitive resources [28].

8.3 Additional Issues

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

First is the issue of application dependencies, that is, how to handle the case when an application embeds an Activity from another application that is not installed. Possibilities include automatically bundling and installing dependencies (as also proposed by the authors of AdSplit [23]), giving the user the option of installing the missing application, or simply failing silently. This issue led the authors of AdDroid [17] to decide against running ads in their own process, but we argue that the security concerns of not doing so outweigh this issue. The concern that users might uninstall or replace ad applications to avoid seeing ads could be addressed by giving parent applications feedback when a requested embedded Activity cannot be displayed; applications relying on ads could then display an error message if the required ad library is not available. Updates and differences in library versions required by apps could be handled by Android by supporting multiple installed versions or simply by the ad libraries themselves.

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

9 Related Work

Finally, we consider additional related work not discussed inline.

In Section 4 and Figure 1, we explored existing implementations of embedded cross-application user interfaces [5, 17, 18, 23]. These systems have differing goals and employ a variety of techniques, but none fully meets the security requirements defined in [18] and expanded here. In particular, none of these approaches can, without modification, support security-sensitive embedded user interfaces like ACGs [19]. The original ACG implementation built on interface-level security properties provided by the Gazelle browser operating system [29].

Others have explored the problem of clickjacking in more depth. One study [20] found that most framebusting techniques are circumventable, making them ineffective for preventing clickjacking. Other work [11] provides a comprehensive study of clickjacking attacks and defenses, presenting a solution (InContext) that relies on the browser to verify the visual context of sensitive UI elements. LayerCake could be extended to support InContext for additional clickjacking protection.

Our implementation relies on security properties provided by the Android WindowManager. Window system security has been explored previously by projects such as Trusted X [4] (an implementation of the X Window System [9] based on the Compartmented Mode Workstation requirements [30]) and the EROS Trusted Window System [22]. We extend this work by leveraging a secure window system to support secure cross-application UI embedding.

10 Conclusion

We have systematically considered the security requirements for embedded user interfaces, analyzing existing systems — including browsers, smartphones, and research systems — with respect to these requirements. While browsers have evolved to address many (though not all) of these requirements over time, Android-based implementations have not supported secure embedded interfaces. We thus created LayerCake, a modified version of the Android framework that supports cross-principal embedded interfaces in a way that meets our security goals. The resulting capabilities enable several important scenarios, including advertisement libraries, Facebook social plugins, and access control gadgets. Based on our exploration and implementation experience, we provide a concrete set of criteria and techniques that has to date been missing for system developers wishing to support secure interface embedding.

This paper, along with any updates, will be available at `https://layercake.cs.washington.edu/`.

11 Acknowledgements

We thank Roxana Geambasu, Alex Moshchuk, Bryan Parno, Helen Wang, and the anonymous reviewers for their valuable feedback on earlier versions. This work is supported in part by the National Science Foundation (Grant CNS-0846065 and a Graduate Research Fellowship under Grant DGE-0718124), by the Defense Advanced Research Projects Agency (under contract FA8750-12-2-0107), and by a Microsoft Research PhD Fellowship.

References

[1] BARTH, A. Security in Depth: HTML5's @sandbox, 2010. http://blog.chromium.org/2010/05/security-in-depth-html5s-sandbox.html.

[2] BEGEMANN, O. Remote View Controllers in iOS 6, Oct. 2012. http://oleb.net/blog/2012/02/what-ios-should-learn-from-android-and-windows-8/.

[3] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium* (2011).

[4] EPSTEIN, J., MCHUGH, J., AND PASCALE, R. Evolution of a Trusted B3 Window System Prototype. In *IEEE Symposium on Security and Privacy* (1992).

[5] ETTRICH, M., AND TAYLOR, O. XEmbed Protocol Specification, 2002. http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html.

[6] FACEBOOK. Social Plugins. https://developers.facebook.com/docs/plugins/.

[7] FACEBOOK. Like button requires confirm step, 2012. https://developers.facebook.com/bugs/412902132095994/.

[8] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: user attention, comprehension, and behavior. In *8th Symposium on Usable Privacy and Security* (2012).

[9] GETTYS, J., AND PACKARD, K. The X Window System. *ACM Transactions on Graphics 5* (1986), 79–109.

[10] GOOGLE. AdMob Ads SDK. https://developers.google.com/mobile-ads-sdk/.

[11] HUANG, L.-S., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: Attacks and Defenses. In *21st USENIX Security Symposium* (2012).

[12] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on WebView in the Android system. In *27th Annual Computer Security Applications Conference* (2011).

[13] LUO, T., JIN, X., ANANTHANARAYANAN, A., AND DU, W. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *5th International Symposium on Foundations and Practice of Security* (2012).

[14] MICROSOFT. User Account Control. microsoft.com/en-us/library/windows/desktop/aa511445.aspx.

[15] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., AND LEVY, H. M. SpyProxy: Execution-Based Detection of Malicious Web Content. In *16th USENIX Security Symposium* (2007).

[16] 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 and Security* (2010).

[17] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)* (2012).

[18] ROESNER, F., FOGARTY, J., AND KOHNO, T. User Interface Toolkit Mechanisms for Securing Interface Elements. In *25th ACM Symposium on User Interface Software and Technology* (2012).

[19] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE Symposium on Security and Privacy* (2012).

[20] RYDSTEDT, G., BURSZTEIN, E., BONEH, D., AND JACKSON, C. Busting Frame Busting: A Study of Clickjacking Vulnerabilities on Popular Sites. In *IEEE Workshop on Web 2.0 Security and Privacy* (2010).

[21] SCHECHTER, S., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The Emperor's New Security Indicators. In *IEEE Symposium on Security and Privacy* (2007).

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

[23] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. AdSplit: Separating Smartphone Advertising from Applications. In *21st USENIX Security Symposium* (2012).

[24] SOPHOS LABS. Facebook Worm: Likejacking, 2010. http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/.

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

[26] TEMPLEMAN, R., RAHMAN, Z., CRANDALL, D. J., AND KAPADIA, A. Placeraider: Virtual theft in physical spaces with smartphones. *CoRR abs/1209.5982* (2012).

[27] W3C. Same Origin Policy. http://www.w3.org/Security/wiki/Same_Origin_Policy.

[28] W3C. Device API Working Group, 2011. http://www.w3.org/2009/dap/.

[29] 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).

[30] WOODWARD, J. P. L. Security Requirements for System High and Compartmented Mode Workstations. Tech. Rep. MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as DDS-2600-5502-87), The MITRE Corporation, Nov. 1987.

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