User Interface Toolkit Mechanisms for Securing Interface Elements

Franziska Roesner, James Fogarty, Tadayoshi Kohno
Computer Science & Engineering
DUB Group, Security and Privacy Research Lab
University of Washington
{franzi, jfogarty, yoshi}@cs.washington.edu

ABSTRACT
User interface toolkit research has traditionally assumed that developers have full control of an interface. This assumption is challenged by the mashup nature of many modern interfaces, in which different portions of a single interface are implemented by multiple, potentially mutually distrusting developers (e.g., an Android application embedding a third-party advertisement). We propose considering security as a primary goal for user interface toolkits. We motivate the need for security at this level by examining today's mashup scenarios, in which security and interface flexibility are not simultaneously achieved. We describe a security-aware user interface toolkit architecture that secures interface elements while providing developers with the flexibility and expressivity traditionally desired in a user interface toolkit. By challenging trust assumptions inherent in existing approaches, this architecture effectively addresses important interface-level security concerns.

ACM Classification: H.5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces.

General terms: Security; Human Factors; Design.

Keywords: Security; user interface toolkits.

INTRODUCTION AND MOTIVATION
User interface toolkits help to reduce barriers to the design and development of modern graphical interfaces [24, 25]. In aiming to provide maximal flexibility and expressivity, existing toolkit research generally makes an implicit assumption that developers have full control of an interface (e.g., [6, 7, 8, 17, 24]). However, as applications move towards interfaces composed of elements from different sources, this assumption can pose significant security risks.

Consider the mashup nature of many modern interfaces, wherein multiple elements of an interface are implemented by different developers with varying trust relationships. Such mashups arise both on the Web and within applications. In a common scenario, a mobile application imports a library to display advertisements in a designated portion of its interface. The application must trust that the advertising library will not abuse the application's permissions without user consent (e.g., to access user information or to send a premium-rate SMS). Conversely, the advertising library must trust that the application will not programmatically click its advertisements in order to increase its advertising revenue. In both cases, this trust can be misplaced: Android ad libraries can and have abused the permissions of embedding applications [10, 13] and Android apps can programmatically click on embedded ads. Malicious programmers can also trick users into clicking a sensitive embedded element by manipulating its display, an attack known as clickjacking [14]. For example, a website can trick a user into clicking a Facebook "Like" button by making the element transparent or uncovering it just as the user clicks in a predictable location.

Such risks traditionally go unaddressed or are mitigated in ways that come at the expense of interface flexibility and usability. For example, a conventional solution for preventing an ad library from illicitly accessing a user's location is to insert a system-controlled confirmation dialog when an application requests location information. Although this approach increases security, it can significantly impact the usability of legitimate applications. Consider also guidelines for security-critical interfaces, such as the Microsoft Windows User Account Control (UAC) guidelines specifying that a shield icon be displayed on buttons that result in actions requiring administrative privileges. In current tools, such guidelines cannot be programmatically enforced but must instead be verified through inspection and review (e.g., an app store process).

Existing mitigations for interface-level security concerns are implemented at the system level, not in the user interface toolkit. We argue that this design is a principal reason that these solutions negatively impact interface usability and flexibility. For example, because traditional user interface toolkits do not support securely embedding a sensitive element into another developer's context, system developers are forced to resort to secure system prompts.

UIST'12, October 7-10, 2012, Cambridge, MA, USA.

We therefore propose that such threats be addressed by rearchitecting the user interface toolkit. A well-designed security-aware user interface toolkit could enforce visual security indicators (such as the UAC shield), link a user's actions in an interface to an application's access to sensitive resources (such as location), and strike at the core of security threats that leverage interface manipulation (such as clickjacking). In addition, we believe that securing sensitive interface elements will enable future innovation in the design of interfaces for security-sensitive interactions.

In this work, we consider security as a primary goal in the user interface toolkit. We explore the security assumptions of existing approaches and develop mechanisms that isolate mutually distrusting interface elements while retaining the flexibility that developers enjoy in existing toolkits. Specifically, this paper contributes:

- Scenarios that motivate a security-aware user interface toolkit for interfaces with mutually distrusting elements.
- A set of security properties needed for these scenarios.
- A security-aware user interface toolkit architecture that achieves these system-level security properties while maintaining necessary flexibility for developers.
- Prototype implementations in the context of mobile application (Android) and Web (browser) toolkits.

SCENARIOS AND SECURITY PROPERTIES

In this section, we introduce a set of scenarios motivating a need for security as a primary goal in user interface toolkits and explore the drawbacks of existing approaches to achieving security in these scenarios. We then consider our threat model and extract a set of desired security properties.

Scenarios

Resource Access. Modern systems (e.g., smartphones, browsers, and desktop operating systems) provide applications with access to a variety of system resources (e.g., a camera, user location, a contact list). Systems protect these resources by granting applications access to them only if permitted by the user, who agrees either to a prompt at the time of first access or to a list of permissions requested at the time of installation (known as a manifest). However, these approaches can pose security risks: after an application is granted access to a resource, most systems allow it to continue accessing the resource whenever and however it wishes, even if that access is invisible to the user and/or inappropriate [29]. For example, an app could take photos or send out location information without a user's consent or knowledge.

Existing work on user-driven access control [29] considers using a person's interactions with an interface for resource access control decisions. For example, a person's click on an embedded location button implies the intent to grant the application location access (see Figure 1). By capturing this intent, the system can ensure that the application accesses location only at the appropriate time, when access is expected by the user. However, this solution assumes that applications cannot trick people into clicking on such

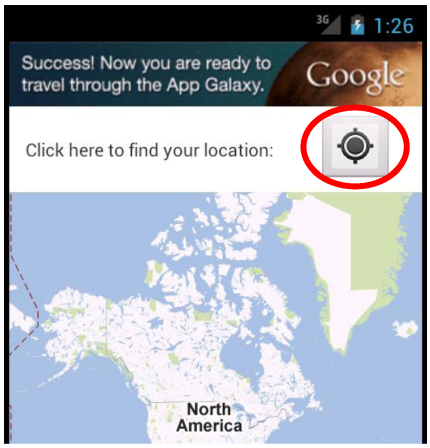

Figure 1: This maps application includes an advertisement generated by an ad library (top) and a system-provided location button (circled).

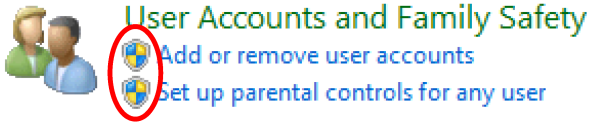

Figure 2: The User Account Control shield in Microsoft Windows (circled) signifies that the resulting action requires administrator privileges.

buttons and cannot manipulate a button's display (e.g., to hide the location icon). Because existing user interface toolkits do not support security for embedded elements, system buttons cannot be embedded in today's interfaces.

Enforcing Interface Requirements. More generally, the system may wish to enforce interface restrictions on elements that can result in restricted operations when users interact with them. For example, Microsoft Windows UAC guidelines [21] state that buttons which will result in actions requiring administrator privileges should be adorned with a UAC shield icon (see Figure 2). As another example, browsers may choose to display visual indicators when users perform insecure actions (e.g., a crossed-out lock icon displayed when using a non-HTTPS connection). If a system wants to require such a visual indicator for all applications making network connections, current tools cannot help in enforcing the requirement. Instead, such a requirement can be enforced only with interface guidelines and application review (e.g., an app store process).

Third-Party Libraries and Transactions. As described in the introduction, applications that embed third-party elements (such as advertising libraries) can present or expose themselves to security risks. Similar concerns arise when applications embed other third-party elements, such as payment elements (e.g., for PayPal), social widgets (e.g., the Facebook "Like" button), or federated login systems (e.g., using a Google account to login to another website).

User interface toolkits currently do not support security in these scenarios. For example, Android applications can easily manipulate interface elements provided by third-party libraries. An application seeking to fraudulently

increase its advertising revenue can, for instance, programmatically click on the ads that it embeds. We have implemented a proof-of-concept Android application demonstrating that three separate ad libraries are vulnerable to this type of attack. On the other hand, existing research [10, 13] indicates that some ad libraries inappropriately exploit the permissions of the embedding application (e.g., stealing contact information and tracking location).

Current approaches to securing third-party interface elements embedded in an application restrict the flexibility of interface designers. Consider the popular Facebook "Like" button, which can be embedded by websites as an iframe to allow visitors to "Like" the embedding page and share it on their Facebook profile. Although the Web's same-origin policy [33] prevents the embedding webpage from programmatically clicking on or manipulating the button's iframe, pages can mount clickjacking attacks in which they trick users into clicking on the button (e.g., by uncovering the button just before the user clicks in a predictable location). When Facebook suspects that a particular button is victim to such an attack, it switches that button into a secure mode [11]. In this mode, clicking on the button opens a popup window in which the user must confirm the action. This is clearly a more cumbersome user experience not desired by Facebook's developers.

In these and other scenarios, there may be expected and mutually beneficial interaction between an embedded element and an application (e.g., informing the application when an ad fails to load or when a payment has been completed). It would thus be too restrictive to completely isolate interface elements from different sources.

Threat Model

In this work, we are concerned with protecting interface elements in one part of an application (e.g., the main application itself) from another part of the application (e.g., an included ad library), and vice versa. We consider developers of these different application components to be potentially mutually distrusting. We further consider any application developer to be a potential adversary or accidental adversary of the system. An intentional adversary might attempt to fool a user or attempt to access sensitive resources without legitimate permission; an accidental adversary might fail to adhere to interface guidelines intended to increase system security (e.g., failing to display the UAC shield) or might accidentally expose a user's data to a third party. We make the standard assumption that the system itself is trustworthy and uncompromised (e.g., the operating system or the browser). In selected cases, where noted, we also rely on a trusted app store review process, though we design our toolkit architecture to minimize this reliance. Later sections also discuss additional limitations of our prototypes on two current platforms (i.e., Android and a Web browser).

Desired Security Properties

To refer to code belonging to different parts of an application or the system, we introduce the term *trust group*. We extract from the presented scenarios and threat model the following security properties needed to protect interface elements within mutually distrusting trust groups.

1. *Display Integrity*: Code in one trust group should not be able to alter the content or appearance of an interface element in another trust group, except via APIs explicitly exposed by that element.
2. *Input Integrity*: Code in one trust group should not be able to programmatically interact with interface elements in another trust group (e.g., calling a `performClick()` function), except when explicitly permitted.
3. *Intent Integrity*: Code in one trust group should not be able to mount clickjacking attacks to force or trick users into interacting with an interface element in another trust group. Code should also not be able to prevent intended user interactions with interface elements in another trust group (i.e., a denial-of-service attack).
4. *Data Isolation*: Code in one trust group should not be able to read or extract content displayed by an interface element in another trust group, except following an explicit user interaction within the element that permits this (e.g., pressing "Select" in a file picking menu would allow the application to access the file system and open the selected file). Code in one trust group should not be able to eavesdrop on input intended for a interface element in another trust group.
5. *UI-to-API Links*: It should be possible for application or system APIs to be linked to interface elements. For example, a `takePhoto()` API might be accessible only from a camera button in the system's trust group.

In designing a security-aware user interface toolkit, we aim to support these security properties alongside the traditional goals of minimizing the difficulty of implementing typical interfaces (i.e., minimizing a toolkit's threshold [24]) while still providing maximal expressiveness and flexibility (i.e., maximizing a toolkit's ceiling [24]).

ARCHITECTING A TOOLKIT FOR SECURITY

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

Trust Groups and Permissions

Our architecture separates mutually distrusting application components, both interface code and application code, into distinct trust groups. For example, system code belongs to one trust group (referred to as "system-trusted" throughout this paper), an application belongs to another trust group, and third-party elements belong to their own trust groups (e.g., an ad library, a PayPal payment button, or a Google login field). Figure 3 shows how parts of an application

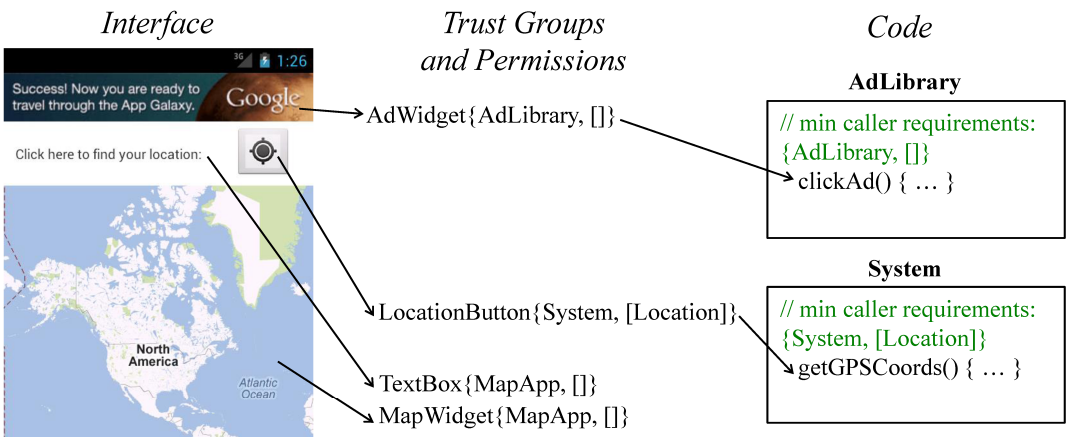

Figure 3: An example of how an element's trust group and permissions map to accessible APIs. An element with trust group A and permissions B and C is designated ElementName{A, [B, C]}. This screenshot is from the Android implementation of our architecture.

might map to trust groups, and we explain the details of this figure throughout this subsection.

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

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

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

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

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

In this paper, we designate an interface element with trust group A and permissions B and C as Element{A, [B, C]}. Figure 3 shows an example of how elements map to trust groups and permissions, which in turn map to accessible APIs in application, system, or library code. In the common case (such as the AdWidget in Figure 3), an element's permissions list will be empty, giving it access to only those APIs that do not require any additional permissions.

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

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

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

- Code cannot manipulate the display of elements in other trust groups (Display Integrity). For example, the methods `setBackground()` or `setTransparency()` can be inaccessible to code from another trust group.

- Code cannot extract data from elements in other trust groups (Data Isolation). For example, the method `getText()` can be inaccessible.
- Code cannot programmatically click on elements in other trust groups (Input Integrity).
- Code cannot enable or disable elements in other trust groups (Intent Integrity).

Another aspect of Intent Integrity is the ability to prevent clickjacking attacks, in which a malicious parent reveals a sensitive interface element from another trust group just as the user is about to click in a predictable location. An element can use system-provided information about its current visibility to implement a clickjacking-protection policy, such as becoming enabled only after being fully visible for some time (as in Chrome [3] and Firefox [23]).

Interface Layout Tree

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

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

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

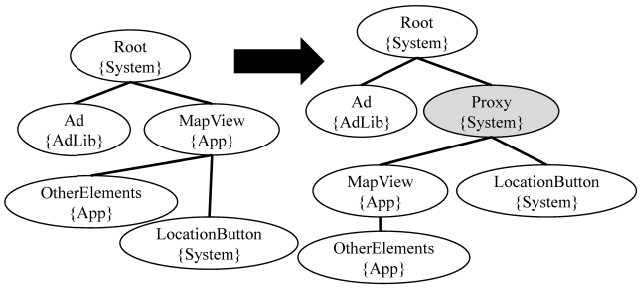

Figure 4: The intended layout tree for the application shown in Figure 1 is transformed to include a proxy node, enforcing the invariant that only system-trusted nodes may have children of a different trust group.

Conflicts in Visual Space. Although our invariants dictate that only system nodes may have children from another trust group, applications may wish to visually embed elements from different trust groups. Consider the example in Figure 1, where an interface consists of an application map element containing a system-trusted location button. The first invariant is easy to satisfy by placing the entire application in a system-trusted frame. However, the second invariant is more difficult to satisfy while achieving the desired visual effect, as it prevents the map element from being the parent of the location button.

We solve this problem by introducing a system-trusted *proxy node* into the layout tree. Figure 4 shows the resulting transformation of the tree. In particular, the proxy node becomes a parent of both the map element and the location button, overlaying them as intended by the application. This transformation preserves the trust hierarchy for layout and for event propagation. We describe in later sections how this transformation can be implemented automatically and transparently to applications, which can continue to manipulate a view of the layout tree that matches the intended visual layout.

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

In the case of a system-trusted child element (e.g., a location button), we rely on our assumption that the system

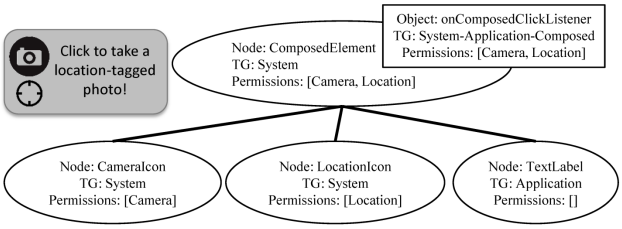

Figure 5: In this example, the application has composed the system-trusted camera and location icons with its own label to create a custom button that has both camera and location permissions.

is trustworthy. Our architecture enforces the minimum requested size for system-trusted elements, even if the size allocated by the parent is smaller than this minimum.

In the general case, we cannot make assumptions about whether the embedding or the embedded node is more trustworthy. We resolve the conflict as follows: if the child element requests a larger size than permitted by the parent element, we draw the child element with the smaller size, visually indicate to the user that the child could not be fully displayed, and allow the user to manually maximize it. This solution can be seen as a sort of generalization of the behavior of typical browser popup-blockers.

Although isolating elements into trust groups and maintaining the described tree invariants supports our desired security properties, naïvely implementing these mechanisms introduces undesired restrictions on interface flexibility. To restore this necessary flexibility, the next sections extend this basic architecture to allow developers to expose model-level APIs, compose elements, and display feedback across trust groups.

Model-Level Event Listeners

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

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

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

Composition Across Trust Groups

We have thus far considered only the *visual* composition of elements belonging to different trust groups. However, interface designers require greater flexibility to *logically* compose elements. For example, consider the Windows UAC scenario in which an arbitrary developer-defined button that includes a UAC shield must be able to access privileged system APIs. To access these APIs, the button must be in the system's trust group. However, its visual layout must be largely defined by the app developer. Conceptually, the developer would like to embed the system-trusted UAC shield element in a custom element and inherit the former's trust group and permissions.

We support composition of elements from multiple trust groups by introducing a system-defined ComposedElement. Any trust group may choose to expose composable sub-elements for inclusion in a ComposedElement (e.g., the system may expose a UAC shield icon with the "administrator" permission or a GPS icon with the "location" permission). Another developer can then mash up these sub-elements with custom elements in a ComposedElement. For instance, Figure 5 shows a composed button for taking a location-tagged photo.

Our toolkit must ensure that allowing compositions does not violate the security properties achieved in previous sections. In particular, a ComposedElement must not allow code to manipulate or observe elements in another trust group or to inappropriately access restricted APIs.

In order to retain the benefits of the layout tree invariants previously described, we assign ComposedElements to the "system" trust group. ComposedElements are thereby allowed to contain sub-elements of other trust groups. Our invariants continue to hold within the ComposedElement. This ensures, for example, that an application cannot use a composition to eavesdrop on a third-party login field.

Recall the goal of a composition is for the resulting element to inherit the permissions of its constituent elements. However, we do not wish to allow applications to inject arbitrary code into a ComposedElement, as this code will run in the system's trust group. To achieve these goals simultaneously, the ComposedElement allows the trust group that embeds it to attach event listeners. Unlike other event listeners (which run in the attacher's trust group), listeners of a ComposedElement run in a temporary trust group derived from the ComposedElement's sub-elements;

its set of permissions is the union of their permissions. For example, the ComposedElement in Figure 5 includes location and camera sub-elements, resulting in an `onComposedClickListener` with trust group and permissions defined as {System-Application-Composed, [Location, Camera]}. The attached listener can thus access the necessary system APIs (but not other system APIs).

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

Flexibility of Feedback

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

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

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

Similarly, lenses [2] are overlaid on an interface to display flexible feedback about the underlying elements. For example, a lens over a set of map tiles might magnify the underlying map features or highlight certain cities. However, the elements from which a lens requires information in order to paint itself may not all belong to the lens's trust group. System-trusted lenses can have full access, but supporting arbitrary lenses requires allowing the lens element to show feedback based on elements in other trust groups. As with drag and drop, we wish to support feedback in the lens without granting the lens's trust group full access to the underlying interface elements.

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

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

ANDROID IMPLEMENTATION

We prototyped our user interface toolkit architecture with an implementation for Android. Android currently shows only one application interface at a time, with all elements in that interface run in the same process. This can include elements defined by Android's toolkit, by the application itself, by an embedded library, and in a limited way by another application (via `RemoteView`). The embedding application is thus trusted with and trusts any element it embeds, an assumption that we challenge with this work.

We implemented our approach in Android 4.0.3 (Ice Cream Sandwich) by modifying its user interface toolkit (Java packages `android.view` and `android.widget`).

API Restrictions

The base class of all Android interface elements is the `View` class, which is extended by Android's built-in elements and can be arbitrarily extended by developers. We modified all methods in this class to take an additional parameter that specifies the trust group of the caller. This additional parameter is a reference to the calling object (we describe below how we ensure the validity of this parameter and how we extract the correct trust group and permissions list from it). The `View` class and its subclasses can thus use this information to restrict or expose methods based on the appropriate policy. By default, we allow calls only from callers of the same trust group as the `View` itself, as well as from system-trusted objects.

For backwards compatibility with existing Android applications, we did not replace all `View` methods but instead duplicated them and added the additional trust group parameter. An app store review process could ensure new applications use only the new methods (via static

analysis or a trusted compiler that replaces deprecated method calls). A non-backwards compatible solution could simply remove the deprecated methods entirely.

Trust Group Declaration and Enforcement

One of the strengths of our toolkit architecture is that it is independent of how the isolation between trust groups is implemented. The main requirement for a trust group implementation is that it prevents a code component from faking its trust group. Our Android implementation uses Java packages to define trust groups. This approach allows applications to split themselves into trust groups of the appropriate granularity, and it naturally separates libraries into separate trust groups (usually included as .jar files). To prevent an application from faking its trust group as that of a library-defined package, the library should seal its package [26]. Package sealing is not currently enforced by Android's Java classloader; this would need to change in order to securely support trust groups based on packages.

Verifying the trust group of a caller requires access to the calling object. Android applications expose and manipulate their interfaces from within Android `Activity` classes, which are in turn extensions of the `Context` class, which contains global information about the application. We modified the `Context` class to include non-overridable `getTrustGroup()` and `getPermissions()` methods; the former returns the trust group of the enclosing object based on the Java package name. Methods that require a trust group must thus be called from within a `Context`; we can again use static analysis at app store review time to verify that callers pass themselves to such methods (i.e., pass `this`). Methods in the `View` class can thereby be assured that the stated trust group of the caller is correct.

Layout Tree Invariants

We further modified Android's user interface toolkit to enforce the layout tree invariants described in the previous section. We achieve this in a manner that is transparent to applications. In particular, we assign to the "system" trust group the Android-defined root element by which each application's interface is attached to its window. When an application attempts to insert a system element as the child of a non-system parent, our modifications rearrange the layout tree to insert the necessary proxy. In practice, we insert a set of three proxies, so that all elements retain a parent of the `ViewGroup` type that they expect (and for which their layout parameters are specified). Figure 6 shows an overview of this transformation.

To make this transformation transparent to the application, our implementation preserves the expected parent and child references for moved nodes. We introduce a dummy child node (see Figure 6) containing a reference to the real child (which has been moved), and we store the original parent reference in the moved child. We return these references for methods requesting parent or child information. Thus, the application can continue to add, remove, and reference elements as if the layout tree reflects the visual tree.

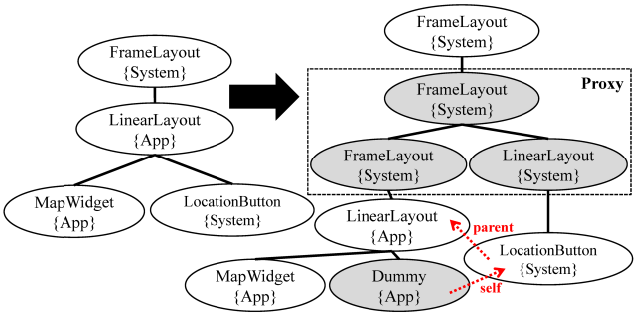

Figure 6: Android layout transformation example. When the LocationButton is added to the tree, two proxy nodes are needed so that both LinearLayout{App} and LocationButton{System} end up with parents of the expected type. The third proxy (a FrameLayout) is used to control the positioning of the other two proxies. The dotted lines indicate pointers tracking intended parent/child relationships.

Flexible Feedback

We modified Android to support flexible feedback for drag-and-drop. We added a new system-trusted element to the toolkit, a `FeedbackView`. When an application wishes to display feedback based on the contents of a passing drag object from another trust group, it creates a new instance of the `FeedbackView` and overlays it upon itself. The `FeedbackView` can thus intercept all drag events intended for the original element.

The application provides the `FeedbackView` a function to generate a bitmap based on drag object data. The `FeedbackView` then provides this function to a sandboxed `FeedbackService` implemented as another Android application with no permissions. The `FeedbackService` generates and returns the appropriate bitmap. The `FeedbackView` displays the resulting feedback bitmap, and the application itself cannot access the contents of the system-trusted `FeedbackView`. The application can thus display drag feedback to a user without gaining access to cross-trust-group drag data.

One limitation of our prototype implementation is that, although the `FeedbackService` runs as an Android application with no permissions, it is not completely isolated. In particular, it may still communicate with other applications via Intents, Android's inter-process communication mechanism. Android currently limits application permissions using manifests. If desired by the Android community, the permission to send or receive Intents could also likely be limited in this way.

Sample Application

We created several sample Android applications to validate our prototype implementation. We summarize one here (shown in Figures 1 and 3), which mashes up an advertisement, a system-trusted location button, and an application's own content. Part of the application interface shows an ad, and another part displays the current location on a map when a user clicks on the location button.

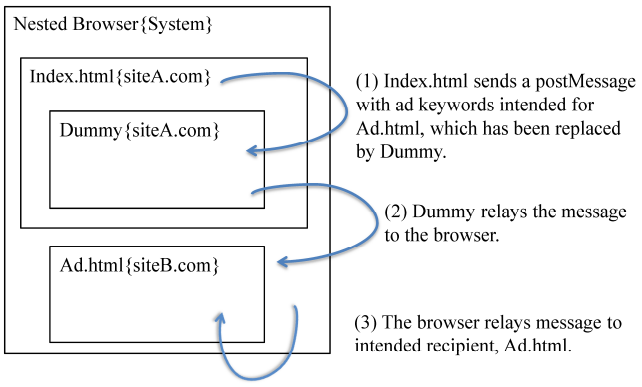

Figure 7: After removing the iframe containing Ad.html from the Index.html page and replacing it with the Dummy, the browser must relay any intended postMessages from Index.html to Ad.html.

The system-trusted location button is a new element that we added to the Android toolkit. It has a fixed "system" trust group and the "location" permission, and it restricts all methods to system-trusted callers, with one exception. It allows any trust group to set a new model-level event listener for location data via `setOnLocationListener()`. Applications can thus embed this button and attach a listener for location data. That data is returned to the application when the user clicks the button. The button also protects itself from clickjacking attacks by responding to input events only while it is completely visible to the user.

The advertisement comes from an ad library embedded by the application. We created a secure ad library by wrapping the existing AdMob library for Android [12]. Our library wraps AdMob's `AdView` in a `SecureAdView`, which sets the `SecureAdView`'s fixed trust group to be the ad library's trust group, thereby restricting all methods to callers in that trust group (or the system group). As a result, our proof-of-concept clickfraud attack for the original AdMob library, in which the embedding application programmatically clicks on the ad, is no longer possible.

WEB IMPLEMENTATION

To further evaluate our approach, we created a prototype implementation for the Firefox Web browser, implemented using iframes and a Firefox add-on.

The Web has long required support for complex embeddings among content from different sources. Web browsers isolate pages and iframes from different origins (based on the same-origin policy [33]). A website containing an iframe has complete control of that iframe's size and the surrounding layout. This control allows the embedding page to, for instance, cover important context and trick the user into clicking on something. Browsers support one-off mechanisms for handling such attacks (e.g., websites can prevent their pages from being framed by another site [30]), but our layout tree invariants ensure that even nested iframes from different origins can securely control their own display properties.

Trust Groups

In our browser implementation, trust groups are defined by standard Web origins (i.e., scheme, domain, and port), which cannot be faked. Interface elements from different trust groups are thus contained within iframes. For example, a webpage in one trust group may embed an advertisement within an iframe from another trust group. As before, an element can have a fixed trust group (defined by its URL) or an owner-bound trust group (defined by the URL of its embedder). These elements use the HTML5 `postMessage` API [15] to expose model-level APIs via cross-origin messages. For example, an advertisement in an iframe from the advertiser's domain might expose a method to set advertising keywords.

Layout Tree Invariants

For simplicity, our current implementation controls the root of all webpages using a browser-within-a-browser (i.e., a trusted webpage that acts as a nested browser, with all navigation done within the nested browser). This simplification allows us to control the root frame of every page (thus satisfying our first layout tree invariant) with minimal changes to the browser itself (see Figure 7).

Our add-on then enforces the second layout tree invariant by modifying pages as they load. In particular, any iframe nested within an iframe of a different origin is removed by the add-on and nested instead within a system-trusted proxy. To preserve the visual effects of the intended nesting, the add-on must move and clip iframes appropriately as their original parent frames are scrolled.

As in the Android implementation, we must retain parent and child pointers expected by the original nodes so that they can continue to communicate with each other via postMessage. We thus replace the moved iframe with a dummy iframe. This dummy, defined by the system but in its embedder's trust group (i.e., owner-bound), relays any postMessages it receives to the moved iframe. This allows the original parent to continue to hold a reference to its visual child, making these modifications transparent to the webpage. Figure 7 shows the sequence of messages sent in the relaying process. Similarly, the browser maintains a mapping of moved iframes to their original parents. If an iframe that has been moved attempts to send a postMessage to its visual parent, the system-trusted actual parent relays this message to the original parent.

DISCUSSION AND RELATED WORK

Considering security as a primary goal in a user interface toolkit presents a unique opportunity to address interface-level security vulnerabilities without sacrificing developer flexibility or expressivity. This approach is a departure from classic security solutions, which operate at a system level or require application review, as well as from conventional user interface toolkits, which have not focused on securing the interaction between multiple mutually-distrusting elements. With the mechanisms that we propose, a security-aware toolkit can ensure that sensitive elements are correctly displayed, that a user's

intended interactions are accurately captured, and that private data is not leaked through interface elements. The remainder of this section examines some related work.

User Interfaces

User interface toolkit research generally focuses on reducing the effort needed to develop new interfaces. Because the assumptions embedded in a framework can limit interface designers and hinder the development or adoption of new techniques, research often focuses on achieving maximal expressivity and flexibility for interface developers [6, 7, 8, 17, 24]. For example, prior work has proposed tools to aid in the creation of mashup interfaces on the Web [16, 19, 36]. However, these toolkits are not aimed at addressing security concerns and their implicit trust assumptions can pose challenges to security.

Work by Arthur and Olsen [1] examined protecting user data by separating interface elements based on trust. They provide a methodology that splits an interface across a trusted private device and an untrusted public device (e.g., a connected screen or keyboard) by surfacing a user's trust choices to applications. Although focused on security, the model is more restricted; we consider a broad set of threats and complex trust relationships within an interface.

Other researchers have focused on designing and evaluating interfaces for security-critical interactions, including phishing protection [9], user account control [22], Web SSL certificate warnings [32], and social network privacy policies [20]. Our work focuses on securing interface-level threats, and we believe that advances here can be leveraged to help enable research in how to best design security-related interfaces.

Security

This work naturally draws upon existing security concepts, such as permissions and capabilities used in operating systems [18]. However, the issue of securing interfaces has not been explored in the security community from the user interface toolkit perspective. Instead, prior security work has focused on isolating applications and content from different sources using system-level techniques (e.g., [4, 28, 34]). These approaches do not consider or are not able to address all of the interface-level threats that we consider (e.g., the authors of [34] do not address a parent element's ability to manipulate the layout of its children).

Our work in this paper draws on user-driven access control [29], which uses system-controlled interface elements called access control gadgets (ACGs) to extract a user's intent to grant system-level permissions to an application (e.g., to access the camera). ACGs, which support only limited composition and customization, represent the most restrictive type of interface elements that we consider in this work. By considering security at the level of the user interface toolkit rather than the system, our proposed architecture subsumes ACGs and supports security properties for a broader and more flexible range of interface elements and scenarios.

It is possible to implement trust groups in ways other than those used in our prototypes. For example, Quire [5] fully separates mutually distrusting code components into different Android applications. To approximate the embedding of interface elements from one app within another (not supported by Android), Quire overlays an application with a transparent background (e.g., the main app) over a second application (e.g., the ad), requiring the top layer to delegate user input events to the bottom layer. To prevent clickfraud, Quire authenticates messages legitimately generated by user input events. In this paper, we explore how a user interface toolkit can leverage trust groups of any implementation to support a wide range of security properties and scenarios.

Although a security-aware user interface toolkit can mitigate a large number of concerns, there remain threats that benefit from alternate mitigation techniques. For example, even with a security-aware toolkit, adversaries can still create fake interface elements to mimic legitimate elements. Fake embedded elements taking user input may be used in phishing attacks (e.g., to steal a password). Sensitive elements can apply existing anti-phishing techniques (e.g., Sitekeys). Although not infallible [31], these techniques can make such attacks more difficult. Additionally, our toolkit can assure that a click on a sensitive element was legitimately intended by the user (e.g., an ad click), but it cannot prevent forged requests to backend servers (e.g., direct requests to the advertising server that mimic the requests made when a user clicks an ad). The Web currently protects against this type of attack (via techniques to prevent cross-site request forgery), but Android does not. Other researchers focus on addressing this type of problem with code attestation [27] or by bringing the Web model to mobile operating systems [35].

CONCLUSION

We have presented a user interface toolkit architecture that maintains developer flexibility while achieving a set of security properties: display integrity, input integrity, intent integrity, data isolation, and UI-to-API links. The mechanisms we propose enable scenarios in which interface elements from different trust groups are embedded in a single interface. In today's interfaces, these scenarios are either insecure (e.g., allowing an application to trick a user into clicking on another trust group's button) or inflexible (e.g., forcing security-related interfaces to be displayed as invasive system prompts).

By isolating code and interface elements into trust groups and by enforcing certain layout tree invariants, our toolkit architecture secures sensitive elements and APIs while still providing developers the necessary flexibility for element composition and for communication and feedback across trust groups. Important questions for future work include the ease with which developers can adopt these mechanisms, as well as support for testing and debugging secure interfaces. In both cases, we note that a security-aware toolkit seems to provide advantages over current ad-

hoc approaches. Our prototype implementations demonstrate that this toolkit architecture is feasible and that important security properties can be enforced with maximal transparency and flexibility for developers.

ACKNOWLEDGEMENTS

We thank Saleema Amershi, Alan Borning, Morgan Dixon, Alex Moshchuk, Kayur Patel, and the anonymous reviewers for valuable feedback on earlier versions of this work. We thank Werner Dietl and Mike Ernst for their insights about Java static analysis. This work was supported in part by the NSF under Graduate Research Fellowship award DGE-0718124 as well as awards CNS-0846065 and IIS-1053868.

REFERENCES

1. R. B. Arthur and D. R. Olsen. Privacy-aware shared UI toolkit for nomadic environments. In *Software Practice and Experience*, 2011.
2. E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *ACM Conference on Computer Graphics (SIGGRAPH)*, 1993.
3. Chromium. Security Issues, Issue 52868. https://code.google.com/p/chromium/issues/list?q=label:Security
4. R. Cox, J. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security & Privacy*, 2006.
5. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, 2011.
6. M. Dixon and J. Fogarty. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.
7. J. R. Eagan, M. Beaudouin-Lafon, and W. E. Mackay. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2011.
8. W. K. Edwards, S. Hudson, J. Marinacci, R. Rodenstein, T. Rodriguez, and I. Smith. Systematic Output Modification in a 2D User Interface Toolkit. In *ACM Symposium on User Interface Software and Technology (UIST)*, 1997.
9. S. Egelman, L. F. Cranor, and J. Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2008.
10. W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
11. Facebook. Like button requires confirm step. http:// developers.facebook.com/bugs/169544703153617
12. Google. AdMob. http://developers.google.com/mobile-ads-sdk/
13. M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.
14. R. Hansen and J. Grossman. Clickjacking. http://www.sectheory.com/clickjacking.htm
15. I. Hickson (ed). HTML5 Web Messaging. W3C Working Draft, 2012. http://www.w3.org/TR/ webmessaging/
16. B. Hartmann, L. Wu, K. Collins, S. R. Klemmer. Programming by a Sample: Rapidly Creating Web Applications with d.mix. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2007.
17. S. Hudson, J. Mankoff, and I. Smith. Extensible Input Handling in the subArctic Toolkit. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2005.
18. H. M. Levy. Capability-Based Computer Systems. Digital Press, 1984.
19. J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-User Programming of Mashups with Vegemite. In *ACM Conference on Intelligent User Interfaces (IUI)*, 2009.
20. H. R. Lipford, J. Watson, M. Whitney, K. Froiland, and R. W. Reeder. Visual vs. Compact: A Comparison of Privacy Policy Interfaces. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.
21. Microsoft. User Account Control. http://msdn.microsoft.com/en-us/library/windows/desktop/ aa511445.aspx
22. S. Motiee, K. Hawkey, and K. Beznosov. Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices. In *ACM Symposium on Usable Privacy and Security (SOUPS)*, 2010.
23. Mozilla Foundation. Known Vulnerabilities, Advisory 2008-08.http://www.mozilla.org/security/known-vulnerabilities/
24. B. Myers, S. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. In *ACM Transactions on Computer-Human Inteaaction (TOCHI)*, 2000.
25. D. R. Olsen, Jr. Evaluating User Interface Systems Research. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2007.
26. Oracle. Sealing Packages within a JAR File. http://docs.oracle.com/javase/tutorial/deployment/jar/sealman.html
27. B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security & Privacy*, 2010.
28. C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *ACM Eurosys*, 2009.
29. F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE Symposium Security & Privacy*, 2012.
30. G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Web 2.0 Security and Privacy (W2SP)*, 2010.
31. S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *IEEE Symposium on Security & Privacy*, 2007.
32. J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX Security Symposium*, 2009.
33. W3C. Same Origin Policy. http://www.w3.org/ Security /wiki/Same_Origin_Policy
34. H. J. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium*, 2009.
35. H. J. Wang, A. Moshchuk, and A. Bush. Convergence of Desktop and Web Applications on a Multi-Service OS. In *USENIX Hot Topics in Security*, 2009.
36. J. Wong and J. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2007.