AUDACIOUS: User-Driven Access Control with Unmodified Operating Systems

Talia Ringer Dan Grossman Franziska Roesner

University of Washington
Seattle, WA, USA
{tringer,djg,franzi}@cs.washington.edu

ABSTRACT

User-driven access control improves the coarse-grained access control of current operating systems (particularly in the mobile space) that provide only all-or-nothing access to a resource such as the camera or the current location. By granting appropriate permissions only in response to explicit user actions (for example, pressing a camera button), user-driven access control better aligns application actions with user expectations. Prior work on user-driven access control has relied in essential ways on operating system (OS) modifications to provide applications with uncompromisable *access control gadgets*, distinguished user interface (UI) elements that can grant access permissions.

This work presents a design, implementation, and evaluation of user-driven access control that works with no OS modifications, thus making deployability and incremental adoption of the model more feasible. We develop (1) a user-level trusted library for access control gadgets, (2) static analyses to prevent malicious creation of UI events, illegal flows of sensitive information, and circumvention of our library, and (3) dynamic analyses to ensure users are not tricked into granting permissions. In addition to providing the original user-driven access control guarantees, we use static information flow to limit *where* results derived from sensitive sources may flow in an application.

Our implementation targets Android applications. We port open-source applications that need interesting resource permissions to use our system. We determine in what ways user-driven access control in general and our implementation in particular are good matches for real applications. We demonstrate that our system is secure against a variety of attacks that malware on Android could otherwise mount.

1. INTRODUCTION

Modern operating systems (such as mobile platforms) isolate applications and limit their privileges. Mobile platforms do not let applications access user resources — such as the camera or location — unless the user grants those permissions to the application. Despite this, applications can still

CCS'16, October 24 - 28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978344

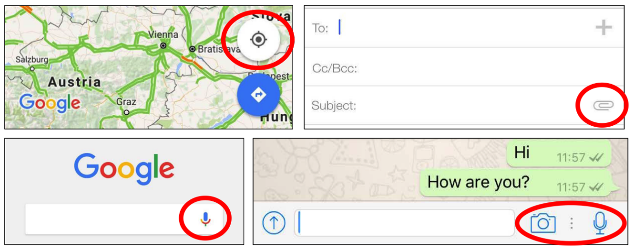

Figure 1: UI elements in applications that, when pressed, cause the application to access sensitive resources. Clockwise from top left: Location in Google Maps, file system in Gmail, camera and microphone in WhatsApp, microphone in Google Search. Modern OSes do not enforce that resources are accessed only through these elements.

steal data or take actions without the user's intention or even knowledge. For example, the FTC recently took action against a flashlight application for leaking user location to advertisers [6], and Android malware is known to covertly send costly premium SMS messages [35].

Mobile platforms typically grant permissions through install-time manifests or runtime prompts. These access control models are problematic: Users find them difficult to understand, and they allow application behaviors that violate user expectations [4, 8, 24, 31]. Prior work introduces *user-driven access control* [24] to improve these models. In user-driven access control, the system extracts permissions information from the user's interactions. *Access control gadgets* (ACGs) [24] are one way of realizing user-driven access control. ACGs are special user interface (UI) elements that let applications access sensitive resources only when the user *naturally* interacts with that portion of the application's UI. Figure 1 shows examples of ACG-like buttons.

User-driven access control is not on a clear path to reaching users. Prior work instantiating ACGs [23, 24] modifies the target OS (Android in the former [23] and ServiceOS [28] in the latter [24]). Changing the OS poses serious deployability challenges. It requires the companies that develop these OSes to make supporting ACGs a priority. Such a drastic change requires significant development effort and impedes backwards compatibility for existing applications. Furthermore, third-party devices often use old OS versions, so there is a long delay before changes reach users [14]. Recent data from the Google Play Store shows that 92.5% of current Android devices run old API versions [1].

Our work makes user-driven access control practical and immediately deployable. We present a design, implementation, and evaluation of user-driven access control that works with no operating system modifications. Our design combines a *secure library* with *static* and *dynamic analyses* to ensure that applications can access sensitive resources only through distinguished UI elements.

We provide the same guarantees as the original user-driven access control design [24]: The application cannot access a **different resource**; the application also cannot access the resource at a **different time**.[1] We move beyond the original design to provide an additional guarantee: The application cannot use the resource for a **different purpose**. Consider a simple camera application: The user expects it to access just the camera, just when the user presses the camera button, and just to save the photo (not to send it over the network). We enforce this without modifying the OS.

In summary, we contribute the following:

1. We enable user-driven access control in entirely *unmodified* operating systems by combining a secure library design with static and dynamic analyses.
2. We instantiate this design in a concrete implementation for Android, which we call AUDACIOUS (Android User-Driven Access Control in Only User Space).
3. We integrate AUDACIOUS into real applications and evaluate how well user-driven access control fits applications in practice.
4. We evaluate the benefits of an approach that requires no OS changes and identify key areas where even limited OS support would be beneficial.

2. CONTEXT

This work targets modern operating systems. These OSes isolate applications and grant them limited permissions. We assume a permission model analogous to the latest Android (API level 23): The OS automatically grants "normal" permissions (for example, network access) at install-time. It grants "dangerous" permissions (for example, the camera) through a prompt to the user at the time of initial use. Once the OS grants permissions, it never revokes them unless the user explicitly revokes them, and it allows the application to use the resources however it wishes. Though we focus our implementation efforts on Android, the broader concepts apply to any modern OS.

2.1 Goals

We aim to support user-driven access control though access control gadgets. Building on the work that introduced ACGs [24], we maintain the first goal:

Goal 1: User-Driven Access Control. Our design should ensure that applications can access sensitive resources only when the user interacts with the corresponding ACG in that application's UI. It should guarantee this even in the face of malicious applications that attempt to circumvent the user or trick the user into interacting with ACGs.

We introduce three additional goals:

Goal 2: Unmodified Operating System. Our design should not make any modifications to the OS.

[1]The original design also guarantees that a **different application** cannot access the resource. We assume an OS that isolates applications, so this is inherently true.

Goal 3: Regulate Resource Use, not Just Access. In the original ACG design, once an application has access to a resource, it may use the resource however it wants. For example, an application may use the user's location both to map the user's run (the expected use) and to send to an advertising server (the unexpected use). Our design should provide guarantees about the flow of resources.

Goal 4: Permission Model Flexibility. User-driven access control is not well-suited for all application scenarios [7]. Some applications may require the ability to access sensitive resources in the background without explicit user interaction. These applications are not necessarily malicious. Consider a calendar application that stays in sync with an online calendar. This application may wish to communicate silently with the calendar over the internet rather than explicitly ask the user to sync the calendar. Our design should allow these applications to use alternative permission models for some of their functionalities.

2.2 Threat Model

We define our system as *sound* if it guarantees for every application that every sensitive resource is accessed as a result of a legitimate user interaction with the expected UI for the ACG and used only as permitted. That is, the application does not access a different resource, access the resource at a different time, or use the resource for a different purpose. We pursue this in the context of the following threat model.

We assume that the OS is trustworthy and uncompromised. We do not modify the OS; instead, we implement a secure library. We assume that our library is implemented correctly. We assume that the static analysis tools and their outputs are trusted. For the sake of simplicity, we accomplish this by adopting the application store model: Users download applications from an application store and the application store runs our analysis tools prior to approval.

Our adversary is a skilled application developer who aims to *improperly access* sensitive resources such as the camera or location. That is, he attempts to access resources in a way that circumvents the ACG-based access restrictions. He may do so by *misusing* the trusted library, *evading* analysis tools, *misleading* the user, or *bypassing* the trusted library.

Our design and implementation assume all code in the application is well-typed Java (we do not support native code). We consider the following classes of attacks out of scope: phishing-style attacks (in which users misidentify applications) and side-channel attacks.

3. TECHNIQUES FOR SECURING ACGS

This section describes our high-level design for securely supporting user-driven access control without OS support. Our design (summarized in Table 1) combines a secure library and program analyses. Applications include our library. In our library, ACGs encapsulate the details of both resource APIs and the UI elements through which users grant permissions to access resources. Our library prevents applications from modifying the ACG UI after it is created and from modifying the ACG validation logic. The library and program analyses together ensure that resource access is authentic (intended by the user) using the techniques we introduce in this section. These techniques occur in three conceptual phases:

Goal	*Technique*	*Sections*
Developer cannot modify the ACG	Library design	§3 and §4.2
Developer can easily include and deploy ACGs	Library design	§3 and §4.2
Events are authentic	Static event analysis	§3.1 and §4.3
Resources, unless specified, are accessed through ACGs	Static information flow analysis	§3.3 and §4.5
Resources, once accessed, are used as expected	Static information flow analysis	§3.3 and §4.5
UI does not deceive the user	Dynamic bitmap check	§3.2 and §4.4
User has enough time to perceive the valid UI	Dynamic bitmap check at random intervals	§3.2.3 and §4.4

Table 1: ACG design overview.

1. **Event Flow** $\text{User} \rightarrow \text{UI}_{\text{seen}}$
   The user (not the application) interacts with the UI.
2. **UI Context** $\text{UI}_{\text{seen}} \simeq \text{UI}_{\text{ACG}}$
   The UI does not trick the user into interacting with it.
3. **Resource Flow** $\text{UI}_{\text{ACG}} \rightarrow \text{ACG} \xrightarrow{*} \dots$
   The application accesses the resource through the ACG and uses it appropriately.

The library combined with these checks form a sound system: An application that passes all three of these checks must use the resource in a way that is consistent with user expectations. Check (1) guarantees that the user genuinely interacts with the ACG UI. Check (2) guarantees that this interaction is deliberate, and that this UI does not mislead the user about the resource that the ACG guards (different resource). Finally, check (3) guarantees that the application does not access the resource when the user does not interact with the ACG (different time) or use the resource in an unexpected way (different purpose).

3.1 Event Flow: $\text{User} \rightarrow \text{UI}_{\text{seen}}$

If the application, not the user, triggers the UI events that access the ACG, then the application impersonates the user. This allows the application to perform arbitrary actions. The flow from the user to the UI must be authentic.

Attack. Eve is developing an evil camera application that secretly takes pictures. She writes code that clicks on the camera ACG button automatically. This is *direct event forgery*: The application creates fake user events.

Eve realizes that this attack may be easy to catch. She disguises the events to make them appear authentic. She adds a misleading button to her application. Whenever the user clicks on this button, she intercepts the event and passes it to the camera ACG. This is *indirect event forgery*: The application uses an authentic event to trigger a forged event.

Defense. In both attacks, the application interferes with the flow of events from the user to the ACG UI. To prevent these attacks, we must ensure that events flow from the user to the ACG UI *without any application interference*. We can accomplish this through a *taint analysis*. Taint analysis tracks the flow of data to some sensitive *sink* and marks anything which passes through a certain *source* as *tainted*. In our case, the sink is the ACG UI code and the source is the application code. That is, we consider events tainted if they pass through application code on the way to an ACG.

More simply and conservatively, we can prohibit applications from constructing and modifying events at all. We suspect the choice of which approach to take should depend on how often benign applications create and modify events.

Time of Defense. We accomplish the event analysis statically. A dynamic approach may incur unnecessary performance overhead since it must occur for every event. This is especially pronounced without OS support; the analysis must determine if an event is tainted rather than rely on an OS-native flag. Since we have enough information for either the conservative check or the taint analysis at compile-time, we prefer a static check.

3.2 UI Context: $\text{UI}_{\text{seen}} \simeq \text{UI}_{\text{ACG}}$

The UI must not trick the user. That is, the actual UI that the user interacts with must be consistent with the UI that we expect for the ACG. Attacks that present UIs to mislead the user are *clickjacking* attacks. The root cause of clickjacking is that UI elements are presented *out-of-context* [12]. We define what it means for the UI to be *in-context* at the time a user interacts with an ACG:

1. **Internal UI** $\text{UI}_{\text{app}} \simeq \text{UI}_{\text{ACG}}$
   The UI for the ACG inside of the application matches the expected UI for the ACG.
2. **External UI** $\text{UI}_{\text{app}} = \text{UI}_{\text{app}} + \text{UI}_{\text{other}}$
   The application UI is not covered by anything outside of the application.
3. **UI Consistency** $\text{UI}_{\text{app}} = \text{UI}_{\text{seen}}$
   The UI we check is the same UI the user perceives.

3.2.1 Internal UI: $\text{UI}_{\text{app}} \simeq \text{UI}_{\text{ACG}}$

The internal application UI must not be deceptive.

Attack. Eve is developing an evil flashlight application that secretly records video. She disguises the video recording ACG as a flashlight button. She covers the video recording ACG button with a small button that says "flashlight." This button hides the intent of the video recording ACG, but still leaves enough room around the edges for the user to accidentally click on it. This is a *cover* attack.

Defense. The UI in the location of the ACG needs to be an *acceptable transformation* of the UI that the ACG expects. The ACG defines the UI it expects and specifies what transformations are acceptable. We check this by comparing the bitmap of the UI to the bitmap that the ACG expects.

In the most basic transformation, the UI must exactly match a predefined UI for the ACG. Often, it is desirable for the transformation to be *stateful*. For example, a button with "on" and "off" states can expect one UI for the "on" state and a different UI for the "off" state. This way, an application cannot trick the user by switching the two states.

An ACG may define a more flexible notion of an acceptable transformation. It may, for example, permit an application to place an element on top of its UI as long as the element does not cover its text. It may call an image-to-text function and ensure that the actual UI has the same text to a user as the ACG UI. It may expose some of its internal UI logic and permit an application to change its colors, but only if the background and the text preserve the same contrast. It may ask the application to provide a proof that the transformation is acceptable.

3.2.2 *External UI:* $\text{UI}_{\text{app}} = \text{UI}_{\text{app}} + \text{UI}_{\text{other}}$

The ACG UI must not be covered by a UI element from a different application or at the system level.

Attack. Eve is developing an evil voice recording application. The user selects a folder and records audio. When the user presses the "stop" button, Eve exposes a system-level notification that says that the audio is done recording, and that the user should click on the notification to save the audio to the folder. Underneath this, Eve places a file deletion ACG button. When the user clicks on the notification to save the audio, the system delegates the event downward and deletes all of the files in that folder instead. This is called *tapjacking* [19, 21].

Defense. We use existing OS features to defend against tapjacking attacks without modifying the OS. We utilize an OS-native flag to determine when the ACG UI is covered by a UI element outside of the application. We reject events on which this flag is set. An alternate approach would be to take a screenshot of the entire UI (including the UI outside of the application) and crop it to the correct location. This may not be allowed by the OS, however, as it introduces security risks: It allows applications to use screenshots to extract sensitive content from each other. Regardless, the library needs some support from the OS to handle tapjacking attacks; we discuss this in Section 5.

3.2.3 *UI Consistency:* $\text{UI}_{\text{app}} = \text{UI}_{\text{seen}}$

The UI we check must be the same as the UI the user perceives, otherwise the check does not protect the user.

Attack. Eve is developing an evil game that collects information about users. She exposes two buttons: an ACG button that toggles location and a button that is part of the game. She places the game button on top. She expects the user to press it at a certain point in the game. Just before this, she switches the location ACG button on top of it. She tricks the user into granting the application permission to access location. This attack is called *bait-and-switch* [12].

Eve isn't sure she will always be able to trick the user into clicking this button, so she saves the event to the location button. She schedules this event to play back to the ACG in the future. This is a *replay* attack.

Defense. The event we check must occur in the *correct location* at the *correct time.* We check that the location of the event is the location of the ACG element, and that the time of the event is the current time minus some buffer.[2]

The user also must have enough time to perceive the UI before interacting with it. The bait-and-switch attack is the result of a UI check that occurs *too late* [12]. The UI check does not occur until the time of the event, but it needs information about the UI at the time the user perceives it *before* the event. We accomplish this through a periodic random check. The check runs randomly within the boundaries of some *check frequency.* If it fails, it invalidates future events to the ACG until some *invalidation interval* passes. As long as this interval is large enough, this guarantees that, at the time of an event, the user has perceived the UI for sufficiently long. If an ACG is already invalid and the check fails, the check extends the amount of time for which it is invalid by the interval; otherwise, it allows the interval to run its course.

We could instead protect against bait-and-switch attacks by running our check in response to changes in layout, but this would be risky: We would then need to prevent applications from intercepting the check to show the expected UI only when the check runs. The check needs to be *unpredictable* to the application developer. The developer may in theory be able to fool the random check by responding to events related to UI rendering; we are unable to implement this attack on Android. If it is possible, we may deter it by introducing a random delay during the check itself.

Choosing the ideal frequency and interval is a trade-off between user experience and security: If we check too frequently, we incur performance overhead, but if we check too infrequently, we risk missing attacks. If we invalidate for too long, then a benign application which changes its UI will provide a poor experience, but if we invalidate for too little time, then the user may not have enough time to perceive the UI. We suspect that the optimal choices depend on the sensitivity of each individual ACG. An ACG which deals with payment methods, for example, will benefit from checking more frequently and invalidating for longer than a voice recording ACG. We evaluate the performance impacts of different frequencies in Section 6.3.

3.2.4 *Time of Defense*

We accomplish the UI context check dynamically. The check must provide guarantees about both *what* UI the application presents and *when* the application presents it relative to any possible user event. A check that only considers *what* UI the application presents may catch an innocent application that modifies the UI at a time that does not deceive the user. A check that only considers *when* the application makes UI changes (a technique used to detect covert application activity [25]) does not address our threat model: A malicious developer can evade the analysis by changing the UI at the right time, but not in a way that is informative to the user. Existing work that accomplishes both of these goals does so through manual image comparison [9, 25].

3.3 Resource Flow: $\text{UI}_{\text{ACG}} \rightarrow \text{ACG} \overset{*}{\rightarrow} \ldots$

The application must not circumvent the ACG and access the resource with no user interaction or in response to interaction with its own UI elements. The application must also use the resource appropriately. This goes beyond the original user-driven access control work, which provides no guarantees on resource use.

Attack. Eve is developing an evil video application. The user expects the application to record and save videos. The user also expects to play his own videos and to play videos on the internet. In reality, the application automatically uploads the user's videos to the internet for all other users to see. This is a *resource flow* attack.

[2]Since we prevent event construction and modification, an application cannot change an event's location or time.

ACG Interfaces	161
Four ACG Implementations	655
ACG UI Logic	249
UI Validation Logic	204
Event Analysis	138

Table 2: Java lines of code in AUDACIOUS, not including existing analysis tools. We implement four ACGs only as needed: one for immediate location access, one for periodic location updates, one for recording audio, and one for playing audio.

Defense. Resources are information, and so the flow of resources is an *information flow* problem. Information flow analysis tracks the flow of program information from sources (for example, user input or the camera) to sinks (for example, the display or the internet). We treat each ACG as a special source or sink. We then ensure that any resource which flows to a given sink is accessed through the ACG. For Eve's application, we can specify that it is legal for a video to flow from the video ACG to the user's filesystem, but that it is illegal for a video to flow from a *different* source (for example, an alternative video recording API) to the user's filesystem.

Viewing this as an information flow problem buys us control: We can specify not only *what* resources must be accessed through ACGs, but also *how* the resources may be used. Eve's application should have both internet and video recording permissions, but it should not send videos over the internet (that is, videos should not flow to the internet). Furthermore, we can verify how resources flow not only *within* an application, but also *between* applications.

This approach also buys us flexibility: In practice, user-driven access control is not sufficient for all modes of interaction with resources [7]. A benign version of Eve's application may wish to automatically sync the user's video feed online without burdening the user with explicit interaction. It can do so by specifying that videos can flow from the internet.

We assume that the underlying information flow tool correctly handles implicit flow, which may leak information from sensitive sources through conditionals [5]. We allow applications to access resources without using our library, but then never use them for anything. These resources cannot be misused by the application.

Time of Defense. We accomplish the information flow analysis statically. This allows the application to provide guarantees on how information flows to the user before releasing the application. This also ensures that there is no negative impact on performance.

4. IMPLEMENTATION: AUDACIOUS

We turn to the implementation of a system for user-driven access control that requires no explicit OS support. We call our system AUDACIOUS: Android User Driven Access Control in Only User Space. The size of our implementation is modest (Table 2).

4.1 System Overview

AUDACIOUS consists of two primary components. The first is a trusted *ACG library* (Section 4.2), which exposes UI elements that provide Android resource access (for example, camera access). Application developers include this library and use its UI elements (for example, camera buttons). The library includes dynamic checks to ensure that application developers do not mislead the user (Section 4.4).

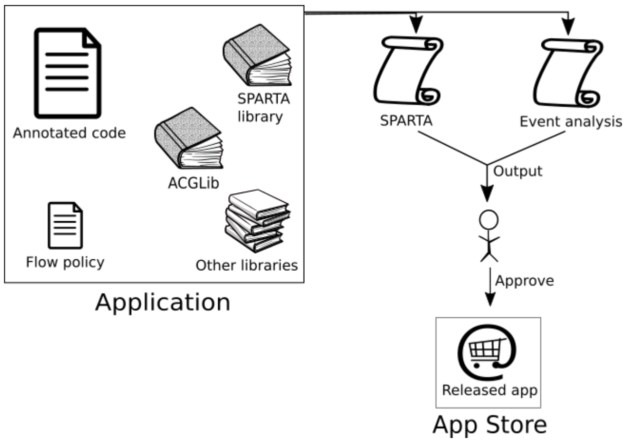

Figure 2: Approval stage. The app includes the ACG library, the SPARTA library, the annotated code, and a *flow policy file*. The annotations help tell SPARTA which flows are present, and the flow policy tells SPARTA which flows are allowable [5]. A human verifier inspects the analysis output.

The second component is a set of static analysis tools which AUDACIOUS uses to verify that application developers correctly include and use the ACG library, and which allow us to support more flexible permission models than strict user-driven access control. This set consists of an event analysis (Section 4.3) and the information flow tool SPARTA [5] (Section 4.5).

These components are combined in two stages. In the *app store approval* stage (Figure 2), the application goes through a combined automatic and manual approval process before it is released in an app store. Our implementation has two static analyses: the event analysis, which runs on the bytecode, and SPARTA, which runs on the source code. Both report output to a human verifier. The verifier investigates any problems and decides whether to release the application. This matches the approval model proposed by SPARTA [5]. In AUDACIOUS, the verifier also inspects any usages of reflection to ensure that the developer does not circumvent the library or analyses.

The app store approval model is just one enforcement method. Alternatively, an organization can incentivize developers to use AUDACIOUS by certifying applications as user-driven access control compliant. We assume the app store approval model for the sake of simplicity.

In the *runtime* stage (Figure 3), AUDACIOUS prevents additional malicious behavior from impacting the user. In particular, it monitors the relationship between *user expectations* and the *runtime behavior* of an application. In practice, this means monitoring the relationship between the *UI* and the *event* that corresponds to a user action. Figure 3 shows three examples of monitoring application behavior.

4.2 Secure Library Design

We implement the ACG library `ACGLib` in Android. An-

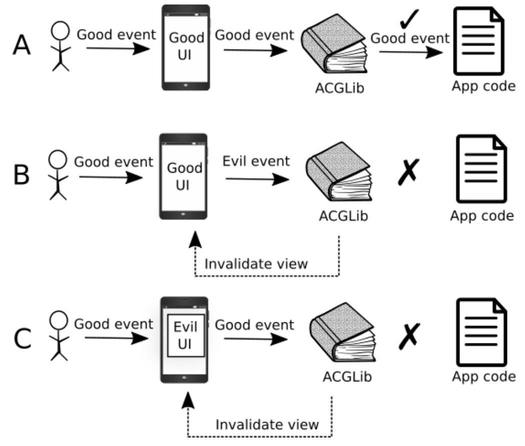

Figure 3: Runtime application monitoring for three scenarios. A) The UI is consistent with the event, B) the event is deceptive, and C) the UI is deceptive. AUDACIOUS accepts only scenario (A).

```
Builder().withRRListener(audio, new RRL() {
    public void onResourceReady() {
        try {
            speaker.passInput(audio.getResource());
        } catch (ACGResourceAccessException e) {
            // error getting file
        }
    }
}).withRRListener(speaker, new RRL() {
    public void onResourceReady() {
        try {
            speaker.getResource();
        } catch (ACGResourceAccessException e) {
            // error playing file
        }
    }
}).build();
```

Figure 4: Chaining ACGs. The `AudioACG` records and saves audio, then passes the recorded file to the `PlayAudioACG`. The `PlayAudioACG` plays the recorded audio to the speaker when the speaker is available.

droid development is *event-driven*: When the user interacts with the UI, Android dispatches events which eventually reach application code and direct behavior. The building blocks of an application are `Activitys`. An `Activity` is a single thing that a user can do; this usually corresponds to a single screen of an application. When an application starts, it executes its main `Activity`, which is associated with a layout. Android exposes a layout's UI elements as `Views`. `Views` are part of a *view hierarchy*, with child `Views` inside of parent `Views`. The developer can change this hierarchy programmatically.

ACGs. We implement each ACG as a `Fragment`. A `Fragment` is an isolated part of an `Activity` (for example, a single button and an action for that button). A `Fragment` contains a `View` that it inserts at runtime. Because of this, each ACG only needs to implement resource-accessing behavior and not any secure UI logic. The application cannot access any of the validation or resource-accessing logic to remove or modify it.

We expose two ACG interfaces: one for *temporary* permissions and one for *permanent* permissions. Temporary ACGs provide resource access only immediately after user interaction, while permanent ACGs provide resource access from the time of initial interaction until the user disables them. In true Android fashion, these are event-driven: The application implements a listener. The listener is notified when the resource availability state changes. When the resource is available, the application may use it as desired.

ACG UIs. We implement a `ViewWrapper` to prevent applications from modifying the ACG UIs after creation. Each ACG UI has an associated `View`. The `ViewWrapper` contains this `View` and does not expose it to the application. It defers all rendering logic and delegates all events to the internal `View`. After the internal `View` handles events, the `ViewWrapper` requests layout. When the internal `View` changes its layout, the `ViewWrapper` notifies Android to draw the `View` again.

Using the Library. `ACGLib` is designed to be easy to use. An application developer can include an ACG in one of two ways: He can define it in a layout and then bind to it from within an `Activity`, or he can add it programmatically at runtime. Next, the developer must implement `ACGListener` and override `buildACGListeners()`, which attaches listeners to the ACGs. The listeners react to changes in resource availability. Developers can also pass input to ACGs and chain ACGs. For example, the ACG for playing audio can take input from the ACG for recording audio (Figure 4).

4.3 Event Flow: User $\rightarrow$ UI$_{\text{seen}}$

Both direct and indirect event forgery attacks can be instantiated two ways in Android: by constructing or modifying events, or by calling the methods that events trigger (such as `performClick()`). In the first case, application interference is application code that runs after setup code before *the event itself* reaches the ACG; in the second case, it is application code that runs after setup code before *the result of the event* reaches the ACG.

We stop applications from constructing or modifying events and from programmatically clicking elements statically. We build our analysis using Soot.[3] Our analysis does not allow application code to call click methods on any subclass of `View`, and it does not let applications call construction, modification, or copying methods on any event subclass. It operates on the bytecode rather than on the source code, since there is enough information present in the bytecode.

We do not stop applications from dispatching events. Event flow through the view hierarchy in Android is complex, so banning all methods which propagate events through the hierarchy is dangerous and does not scale. This does not impact the soundness of AUDACIOUS: Since applications cannot modify, create, or copy events, the only events that applications can dispatch are authentic. This means that at runtime, both the *location* and the *time* of the event are authentic, so we can defer reasoning about these to our UI context check (Section 4.4).

Our implementation is conservative: It guarantees that applications do not interfere with the flow of events from the user to the UI by preventing applications from interfering with events in any way. Some applications modify events for the sake of upscaling or UI customization. We acknowledge

[3]https://sable.github.io/soot/

this as a limitation of our implementation. Our evaluation of event usage in practice (Section 6.4) shows that this behavior occurs rarely enough that it is human-verifiable.

4.4 UI Context: $\mathrm{UI}_{\mathrm{seen}} \simeq \mathrm{UI}_{\mathrm{ACG}}$

Android provides a method that a security-sensitive `View` can override to implement fine-grained security checks for events. We implement our UI context check in this method.

Internal UI: $\mathbf{UI}_{\mathbf{app}} \simeq \mathbf{UI}_{\mathbf{ACG}}$. An application may mislead the user by modifying a `View` or by covering it with a new `View`. To prevent this, each ACG contains a `Bitmap` validator. The validator takes a `View` and decides whether the `Bitmap` rendering of that `View` is acceptable. It is flexible enough to support different notions of acceptable transformations as well as application-supplied proofs. Different ACGs can use different implementations of the validator.

We implement two example validators. The *non-stateful* validator checks for equality between a `Bitmap` rendering of the ACG `View` in isolation and a `Bitmap` rendering of the supplied `View`. This is the default validator, as it is the most conservative. The *stateful* validator takes both a `View` and a state, and expects different UIs for different states. We use this validator for the `LocationACG`, which expects one UI for its "on" state and a different UI for its "off" state.

External UI: $\mathbf{UI}_{\mathbf{app}} = \mathbf{UI}_{\mathbf{app}} + \mathbf{UI}_{\mathbf{other}}$. Tapjacking in Android occurs through `Toast`s. A `Toast` is a system-level pop-up notification. A voice recording application, for example, may use a `Toast` to alert the user that it has saved the audio to the device. Android delegates events which pass through a `Toast` to the UI element underneath it. Malicious applications abuse this feature to trick users into clicking on elements [19, 21]. This is a concern not only within an application, but also between applications, since a `Toast` is outside of the application's view hierarchy.

To prevent these attacks, Android provides the native flag `FLAG_WINDOW_IS_OBSCURED`. This is set on an event whenever the `View` on which the event occurs is either *partially* or *wholly* obscured by a window outside of the view hierarchy, regardless of whether the event passes through the obscured part of the window [3]. We reject an event whenever this flag is set and invalidate the UI for a suitable period of time.

Due to a bug in Android, the flag does not handle partially obscured views correctly. A malicious application can, for example, expose a `Toast` which obfuscates the intent of the ACG, but leave room around the edges. Clicks which pass through those edges ought to be invalidated, but they are not. We have reported this to Android. It is scheduled to be investigated for a future release.

UI Consistency: $\mathbf{UI}_{\mathbf{app}} = \mathbf{UI}_{\mathbf{seen}}$. To ensure that the event occurs in the correct location, we verify that it occurs within the boundaries of the ACG `View` element. To ensure that the event occurs at the correct time, we verify that no amount of time longer than some buffer has passed (that is, the current time is within the boundaries of the time of the event and the time of the event plus the buffer).

We implement random checks to handle bait-and-switch attacks. We expose two intervals at the library level: the maximum frequency of random checks and the invalidation interval for failed checks. Individual ACGs can configure these intervals independently. We set the default time buffer for events to the rate at which random checks occur. This way, a random check is guaranteed to happen in the time an event is valid.

We are once again limited by our lack of OS support in that we are unable to randomly check the UI outside of the view hierarchy. In order to check `FLAG_WINDOW_IS_OBSCURED` without an authentic event, we must construct an event in the location of the ACG. However, if we construct an event programmatically, the flag is never set.

```
public final class LocationACG {
    @Source(value = "ACG(location)")
    public Location getResource();
}

public final class PlayAudioACG {
    public Void getResource();

    @Sink(value = "ACG(play_audio)")
    public void passInput(File file);
}
```

Figure 5: Stubs for two ACGs. The source of any `Location` from the `LocationACG` is `ACG(location)`, and the sink of any `File` passed into the `PlayAudioACG` is `ACG(play_audio)`.

```
ACG("audio_recording") -> FILESYSTEM("/ACG")
FILESYSTEM("/ACG") -> ACG("play_audio")
ACG("audio_recording") -> ACG("play_audio")
```

Figure 6: Flow policy file for a voice recorder application that uses ACGs. The application can save and play its recordings.

4.5 Resource Flow: $\mathrm{UI}_{\mathrm{ACG}} \rightarrow \mathrm{ACG} \xrightarrow{*} \dots$

We use SPARTA [5] to verify flow from the ACG UI to the resource itself and onward. SPARTA is an information flow tool which uses static type-checking to infer illegal flows. SPARTA operates on the source code. Application developers annotate code with *sources* and *sinks*. The annotation burden for developers is typically low [5]. SPARTA includes trusted annotations in the form of *stub files* to minimize annotation burden; this includes most Android core code. SPARTA compares the flows it finds with allowable flows defined in a *flow policy file* (Figure 6). A human verifier inspects the source code and the output of SPARTA in the context of this policy.

SPARTA alone ensures that the application uses resources correctly. However, it does not ensure that the application accesses the correct resources at the correct times. For example, SPARTA will catch an application that sends videos over the internet if the application should never send videos over the internet. However, it will not catch an application that ought to send videos over the internet *with* the user's consent, yet actually sends videos over the internet *without* the user's consent. AUDACIOUS provides more powerful guarantees than SPARTA alone.

We extend SPARTA with the notion of an ACG. We implement an ACG type, which we parameterize by the name of the ACG. This way, SPARTA can distinguish ACGs from other sources and sinks, and it can distinguish different ACGs from each other. We add stub files that inform SPARTA when resources flow to or from an ACG. Adding a new ACG only involves a few lines of stubbing in SPARTA

(Figure 5) and decreases annotation burden for the application developer.

The underlying analysis for SPARTA is sound and correctly handles implicit flow [5]. SPARTA also handles reflection and `Intents` (two major challenges for static analysis of Android applications) soundly [2]. `Intent`s are objects that Android applications use to communicate between parts of an application or between applications. Reflection and `Intent`s complicate control flow. Applications that use these features must annotate their applications accordingly [2].

In AUDACIOUS, applications may use alternative permission models for any functionality for which user-driven access control is not ideal by including non-ACG flows in their flow policies. These applications must have their flow policies approved. Applications that use ACGs without advanced functionality only require manual approval in the case of false positives.

5. DISCUSSION: OS SUPPORT

The previous sections show that supporting user-driven access control *without* explicit OS support is possible, moving beyond prior work on ACGs (e.g., [24]), and enabling ACGs to be used and enforced without requiring major changes to already-deployed OSes. We take a step back and reflect on the benefits of ACGs that don't require OS support, as well as the lessons we learned about the possible role of OS support. We identify areas where limited (not ACG-specific) OS support is necessary or beneficial, and we identify fundamental limitations of not having OS support.

5.1 Benefits of a User Space Approach

An approach that provides and ensures the security of ACGs without OS support has clear benefits.

Deployability. Changes to the OS are costly and may not be backward-compatible. As a result, despite the potential for ACGs to solve many issues with permission models in modern OSes, they are unlikely to be deployed in the near future, if ever. Furthermore, even if ACGs were incorporated into an OS, old devices may never update to the latest OS, and so these changes may never reach customers (a situation notorious for the Android ecosystem [1]).

In contrast, a library paired with program analyses can be deployed within any application and verified by any entity with the ability to run the analysis tools. While we assume that the app store takes on the role of analyst, this role can also be filled by trusted third-party organizations (for example, to produce a list of "ACG-certified" apps).

Flexibility of Permission Models. The application permission model (manifest, prompts, user-driven access control, or some combination) is fixed within an OS. However, permission models are not one-size-fits-all: Different models may be better suited for different applications and different resources [7]. A key benefit of a user space approach like ours is that the permission model can be changed or combined with other permission models. For example, in our implementation, we extend Android with a user-driven access control model, but still allow applications to use the manifest model (subject to information flow guarantees).

5.2 Opportunities for Limited OS Support

Through our design and implementation experience, we identify simple, broadly useful features that OSes can (and may already) provide to enable user-driven access control to be implemented easily and securely in user space. Note that the features we describe here are generic, not ACG-specific. We believe that these features are also useful beyond ACGs (for example, to prevent a broader class of UI-level attacks like clickjacking).

Event Analysis. If the OS prevented events from being created or modified, we would not need to run an analysis to detect illegal event flow. Alternatively, if the OS flagged programmatic events as *synthetic*, we would be able to verify event flow with minimal effort and performance overhead at runtime. Unfortunately, Android does not distinguish between forged events and real events that come from user input, which forces us to make this distinction.[4]

Tapjacking. Detecting tapjacking attacks which occur outside of the application's UI — for example, by another application overlaying content on top of the target application — requires some OS support. The reason for this is that OSes typically do not allow applications to take system-level screenshots, which is necessary for our bitmap inspection approach. (Allowing applications to take system-level screenshots poses a security risk, since it allows applications to read content from other applications.)

Instead, our implementation relies on Android's `FLAG_WINDOW_IS_OBSCURED` flag (modulo its incorrect handling of partially obscured UIs, discussed in Section 4.4). Without this limited support, it would have been difficult or impossible to detect application-external UI attacks.

Bait-and-Switch. With full OS support, we could deter bait-and-switch attacks without relying on random checks. We could instead depend on a secure layer at the OS level which alerts us when the UI changes, but is not susceptible to modification or interception by applications.

5.3 Limitations of a User Space Approach

Finally, we identify several fundamental limitations of supporting user-driven access control without OS support.

Increased Trusted Code Base. We must now trust not only the OS and the device, but also our library, the analysis tools, and the verification model those tools depend on.

Software Updates. If the ACG library is updated, each application developer must individually update their application to include the latest version of the library. By contrast, OS updates (though they may be slow to propagate [1]) are applied centrally.

Inconsistent Permission Models. In some cases, flexible permission models may lead to inconsistent permission models. For example, as of Android 6.0, sensitive permission requests generate permission prompts on first use. If an application accesses that permission through an ACG, this prompt will seem redundant to users (and will be redundant, from a security perspective). Without modifying the OS, however, these prompts cannot be removed even in cases where permission is already granted through an ACG.

[4]Android events do contain a flag called `FLAG_TAINTED` for when the event is inconsistent with previous events. However, the methods to access it are hidden, it is not documented, and we never see it set in any of our attacks.

Application Limitations. Finally, our design and implementation place certain limitations on applications. For example, since unrestricted use of reflection allows an application to circumvent any secure library, we require that developers who use reflection annotate their applications accordingly and that a human verifier inspects any usages of reflection. Though alternate static analysis approaches may remove such restrictions, approaches without OS support are more likely to encounter such challenges.

6. EVALUATIONS

We show that it is feasible to integrate our implementation with real applications. We also test it on malicious code and show that it prevents attacks in the scope of our threat model. We evaluate the performance impacts of different random check frequencies. Finally, we evaluate how often applications create and modify events in practice.

6.1 Aiding Good: Porting Applications

We integrate AUDACIOUS into existing applications to evaluate the development impact. We find the following:

- Some applications have ACG-like UI elements. In these applications, integrating ACGs decreases code size.
- In applications that do not have ACG-like UI elements, integrating ACGs only slightly increases code size.
- Flexibility of permission models is desirable. All of the applications that we evaluate have permission uses for which ACGs may not be ideal.
- Applications benefit from increased customization of and interaction with ACGs. A software-based approach should consider this in its design.
- A proactive developer can decrease the burden of information flow analysis by following best practices.

6.1.1 Applications

We build AUDACIOUS into five open-source applications from the F-Droid repository.[5] We summarize these applications in Table 3. We select these applications for non-trivial uses of permissions and for varying complexity. Simple applications have little advanced functionality; more complex applications may have complicated permissions uses or pass information through `Intent`s.

6.1.2 Development Impact

We evaluate the impact of integrating AUDACIOUS in three places: the library, the event analysis, and the information flow analysis. We count Java LOC using SLOCCount.[6] We summarize our results in Tables 4 and 5.

ACG Library. Integrating the library requires few code changes. Three of the five applications already include unverified UI elements that behave like ACGs. For example, Speed of Sound has a toggle button to start and stop location tracking; this functions like the `UpdateLocationACG`. Integrating the library into these applications decreases code size. Two applications do not have ACG-like UI elements. For example, Solar Compass accesses location automatically when the application starts. Integrating the library into these applications only slightly increases code size.

Event Analysis. The event analysis takes at most 24 seconds and reports no false positives in four out of five applications. It reports two false positives in the `rotateTouchEvent` method of a map class in a dependency of WikiJourney. This method determines the orientation of the map and rotates events accordingly. This is not malicious behavior; the human verifier can check this in the approval stage.

Information Flow. Annotation burden is low for simple applications and higher for more complex applications. The flow for Pinpoi is especially complex, as it allows users to import points of interest from many different data sources and uses `Intent`s heavily. We use a tool in SPARTA to automatically infer annotations for Pinpoi. This tool infers 487 annotations; we correct incorrect annotations and add missing annotations. We expect that this increases our annotation count. Overall, the developer understands the code best and so is in the best position to decrease this burden.

SPARTA takes at most 96 seconds and reports no errors for three of the annotated applications. It reports one error in Speed of Sound and 21 errors in Pinpoi. These are the results of illegal flows. For example, Speed of Sound overrides `toString()` and leaks sensitive information from Bluetooth (BLUETOOTH $\rightarrow$ ANY). Pinpoi catches and propagates runtime exceptions (ANY $\rightarrow$ ANY). Neither applications use these flows maliciously; this is human-verifiable. Nonetheless, SPARTA reports errors to prevent these behaviors. A proactive developer can decrease the number of false-positives by following best practices (for example, by only leaking sensitive information in sensitive methods, or by using checked exceptions).

6.1.3 Lessons and Observations

ACGs encapsulate existing behavior. When applications already have ACG-like UI elements, integrating the library is simply a matter of swapping those elements with ACGs, which *decreases* code size. However, even when applications do not have ACG-like UI elements, integrating the library only *slightly increases* code size. That is, even though we must add new user interaction, we may remove some of the resource-accessing code. Overall, we find that ACGs act not only as a security guarantee for the user, but also as a reusable component for the developer.

Flexibility of permission models is desirable. All five applications have permission uses for which user-driven access control may not be ideal. Consider Speed of Sound: The application saves songs to a database while the user is driving, but the user cannot interact with the application at that time. We can require the user to give permission to save songs before beginning a route or upon finishing a route. However, this may impose an unwanted burden on the user. This supports the claim that ACGs alone do not fully represent all modes of interaction with resources [7]. Furthermore, in practice, it may not be feasible to enumerate all possible uses of ACGs. We can avoid these problems by allowing applications to use alternative permission models for functionalities for which user-driven access control is not ideal.

Applications benefit from increased customization of and interaction with ACGs. The original model for ACGs [24] introduces three types of ACGs: one-time, ses-

[5]https://f-droid.org/

[6]http://www.dwheeler.com/sloccount/

Application	Purpose	Complexity	Resources	ACG Integrated
VoiceRecorder	Record and save audio	Simple	Audio, filesystem	`RecordAudioACG`
Solar Compass	Show position of sun relative to location	Simple	Location	`LocationACG`
Speed of Sound	Adjust volume based on driving speed	Complex	Bluetooth, location, internet	`UpdateLocationACG`
Pinpoi	Manage points of interest	Complex	Location, internet, filesystem	`UpdateLocationACG`
WikiJourney	Show Wikipedia articles for nearby locations	Moderate	Location, internet, filesystem	`UpdateLocationACG`

Table 3: Summary of applications.

	VoiceRecorder	*Solar Compass*	*Speed of Sound*	*Pinpoi*	*WikiJourney*
Original Layout	52	224	338	586	458
Original Java	139	308	1774	3897	1283
Layout Changes	21	7	3	17	7
Java Changes, ACGs	-50	9	-71	-16	28
Java Changes, SPARTA	3	7	79	129	41
SPARTA Annotations, manual	3	14	149	224	178
SPARTA Annotations, automatic	0	0	0	487	0

Table 4: Code changes to integrate ACGs.

sion, and permanent. This continues to adequately model ACG use in applications. However, applications depend on further customization and interaction. A software-based design should consider these use cases. Paradigms we encounter include the following:

1. **Configuration**: Applications pass configuration options to Android resource APIs. For example, VoiceRecorder configures the audio recording API to customize the audio output format.
2. **Communication**: Applications need to know not only when the resource is available, but also when the user interacts with the ACG in other ways. For example, VoiceRecorder starts and stops a timer that displays the duration of the recording, so it needs to react when the user presses the "record" button.
3. **Updates**: Applications request periodic updates from resource APIs. Three out of the five applications that we evaluate ask for location updates when the user's location changes by some amount or when some amount of time passes. This appears to be a common mode of interaction for location access.

6.2 Fighting Evil: Security Analysis

We instantiate the classes of attacks we outline in Section 3 and test that our tool defends against them. We are unaware of available open-source malware for these attacks, so we create our own testing framework for generating application variants. We implement 57 attacks in EvilApp, a test application. AUDACIOUS prevents 55 of these. The two that it does not prevent are limited by lack of OS support. We revisit these attacks and detail the point at which AUDACIOUS prevents them.

UI modification attacks change the ACG UI after creation to trick the user into interacting with the ACG. Library design decisions prevent these attacks: The attacks compile, but have no effect on the UI.

Event forgery attacks create or modify events, or call methods like `performClick()`. Some of these try to trick the analysis by obfuscating the event flow. The static event analysis catches all of these attacks.

Cover attacks add application-level elements that overlap with the ACG UI. The bitmap check stage of the dynamic UI context check catches these.

Tapjacking attacks use system-level pop-up notifications (`Toasts`) to cover the ACG UI and trick the user. The dynamic UI context check catches all but one of these. AUDACIOUS does not catch the attack that *partially* obscures the ACG UI because of the bug we have reported to Android.

Replay attacks capture events from the location of the ACG and schedule them for a later point in time. The UI at the time of the original event may or may not match the ACG UI. The UI consistency stage of our dynamic UI context check catches these.

Bait-and-switch attacks cover the ACG UI and then, at the last minute, put the ACG UI on top so that at the time of the UI check, the UI looks authentic. The random bitmap check stage of the dynamic UI context check catches all but one of these. AUDACIOUS does not catch the attack that uses a `Toast` to do this, since there is no way to check the obscured flag during a random check.

Resource flow attacks circumvent the library: They call resource-accessing APIs directly, access resources that are different from the ones that users expect, or expose their own buttons and directly access resources in response to them. AUDACIOUS catches these attacks statically using SPARTA, as all of them expose invalid flows.

6.3 Random Check Frequency

AUDACIOUS uses random checks to guarantee that the user has enough time to perceive the UI before interacting with an ACG. Smaller frequencies for random checks provide better protection against timing-based attacks. However, frequent random checks add performance overhead. A more sensitive ACG (for example, an ACG that handles payment methods) may wish to use a smaller frequency at some cost to performance.

We evaluate the performance impact on the UI of different validation frequencies on Solar Compass. Over 40 seconds, we toggle the `LocationACG` button 30 times. We compute the total time from the beginning of the first event to the

	VoiceRecorder	*Solar Compass*	*Speed of Sound*	*Pinpoi*	*WikiJourney*
SPARTA Warnings	2	1	8	26	17
SPARTA Errors	0	0	1	21	0
Event Analysis Errors	0	0	0	0	2
SPARTA Time (s)	46	55	77	96	66
Event Analysis Time (s)	5	9	7	9	24

Table 5: Static analysis performance.

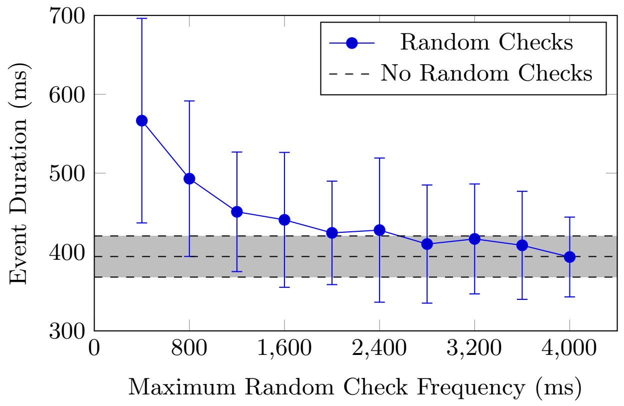

Figure 7: UI performance of Solar Compass. At 400 milliseconds, checks and events coincide so frequently that performance is significantly impaired. At 4000 milliseconds, checks and events rarely coincide, so there is minimal impact on performance.

end of UI rendering for each toggle action for maximum frequencies ranging from 400 milliseconds to 4000 milliseconds. Note that these frequencies are *maxima*: At 400 milliseconds, the random check occurs at a frequency randomly distributed between 0 milliseconds and 400 milliseconds (on average, 200 milliseconds). We compare this to a control version of Solar Compass with random checks disabled. We compare the results in Figure 7.

Overall, we find that the checks are only harmful to UI performance when they occur at the same time as user interaction. This causes a delay in processing the event and rendering the UI. This happens more often for small frequencies, which results in a larger variance of event times and a larger mean. This becomes less likely as the frequency increases, leading to better performance for larger frequencies.

6.4 Event Forgery in Real Applications

Applications sometimes create or modify events or clicks for the sake of UI customization or upscaling. We run our event analysis on the top 100 free applications in the Android application store to determine how often this happens.

In total, our analysis finds 1060 errors across 88 of the top 100 applications (10.6 on average). Of these, 218 are from advertisement libraries or their dependencies. Most errors in these libraries appear related to unskippable video advertisements and interstitial advertisements which may violate user expectations. These errors require manual inspection. We expect non-free applications to have fewer instances of event and click forgery due to the lack of advertisements.

7. RELATED WORK

Prior work documents the shortcoming of permission models, particularly for Android [4, 8, 24, 31]. Our work builds on the notions of user-driven access control and ACGs, introduced by those names by Roesner et al. [24] but more generally dating back to concepts like the powerbox [17, 27] for secure file picking. ACGs for Android are supported by LayerCake [23], requiring significant modifications to Android. In contrast, AUDACIOUS securely supports ACGs for unmodified OSes.

Overhaul [18], like AUDACIOUS, addresses the difficulty of deploying user-driven access control and implements a variant into existing OSes retroactively. Other works explore ideas similar to user-driven access control, including Gyrus [13], AppIntent [33], and the EROS Trusted Window System [26].

Felt et al. [7] argue that secure UIs should be combined with other permission granting approaches depending on the permission type. In that spirit, by combining ACGs with information flow, AUDACIOUS supports both user-driven access control *and* install-time manifests for flows that are not well-suited for ACGs in a particular app's context.

Recent work addresses unwanted resource flow in Android. SPARTA [5], which AUDACIOUS leverages, statically verifies that applications use only those information flows declared by the developer in a policy file. TaintDroid [4] dynamically tracks information flows in Android. AppFence [9] utilizes TaintDroid to introduce privacy controls that allow users to withhold data from applications and prohibit select resources from flowing to the network. These tools provide no guarantees that applications access the correct resources at the correct times based on user interaction.

ClickRelease [16] uses symbolic execution to verify policies that constrain the usage of resources based on user interaction with the application. Rubin et al. [25] statically detect covert communication in Android applications. Unlike AUDACIOUS, these works do not enforce any guarantees on the appearance of the UI at the time of user interaction.

AsDroid [11] detects stealthy resource access by using program and text analysis to determine whether program behavior matches the user's expectations. Though similar in spirit, AsDroid handles only a subset of the issues AUDACIOUS does. Our techniques to handle dynamic UIs and programmatic clicks could be applied to improve AsDroid.

Jekyll on iOS [29] details attacks that pass application approval, but introduce illegal information flows after installation. These attacks rely on violations of memory and type safety (buffer overflows and incorrect type casts) for which we are unaware of attacks in Java's managed environment. As AUDACIOUS does not support native code, these attacks are not relevant to AUDACIOUS.

Other relevant works include SUPOR [10], which statically detects leakage of sensitive user inputs; AutoCog [22] and WHYPER [20], which assess permission-to-description fidelity; and Quire [3], which dynamically provides provenance for on-device IPCs. Many other works have also used program analysis to detect malicious behaviors in Android applications (e.g., [30, 32, 34]).

8. CONCLUSIONS AND FUTURE WORK

Previous work on user-driven access control relies on major OS modifications. This is a barrier to its deployability. We design a system for user-driven access control *without* modifying the OS. Our design combines a secure library with program analysis to ensure that applications use ACGs correctly. Our approach enables applications to combine user-driven access control with other access control models. We demonstrate that this approach can handle most classes of attacks with no OS support. Still, we find that limited OS support is particularly beneficial in preventing some attacks.

We implement our design in Android and integrate it with existing applications. In doing so, we identify techniques to minimize the analysis burden for developers as well as design considerations for future libraries. Our evaluation shows that many applications already use ACG-like UI elements, and that for those applications, integrating the library actually *decreases* code size. Furthermore, even the simplest applications benefit from flexible permission models.

Future Work. Future work may assess acceptability of a UI transformation statically; this would benefit existing works that rely on manual image comparison [9, 25]. A large-scale evaluation of the ways in which applications interact with resource APIs and ACG-like UI elements may better inform future design decisions. In future design, rather than rely on a secure library, we may provide developers with the option to mark existing UI elements and verify their behaviors. Future work applying our design to desktop OSes may need to consider that random checks are insufficient for some attacks involving mouse pointers [12]. A future implementation may repurpose FlowTwist [15], a static taint analysis for Java, as an alternate to the conservative event analysis; the primary barrier is in getting it to work with recent Android APIs. A future implementation may extend ACGs and the event analysis to cover accessibility and voice interactions; the use of `Fragment`s makes this possible with few changes. Finally, we leave securely relaxing the restrictions AUDACIOUS places on ACG UIs to future work.

9. ACKNOWLEDGEMENTS

We thank Michael Ernst and Suzanne Millstein for help integrating SPARTA into AUDACIOUS. We thank Julia Rubin, Kris Micinski, and Zach Tatlock for the stimulating discussions about acceptable UI transformations. We thank all of the students in the UW PLSE and Security labs who provided feedback on early revisions.

References

[1] Android. Dashboards. http://developer.android.com/about/dashboards/index.html, 2016. Accessed: 2016-05-03.

[2] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *30th International Conference on Automated Software Engineering*, 2015.

[3] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Conference on Security*, 2011.

[4] W. Enck, P. Gilbert, B.-G. 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 *9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[5] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *ACM Conference on Computer and Communications Security*, 2014.

[6] Federal Trade Commission. Android flashlight app developer settles FTC charges it deceived consumers, Dec. 2013. https://www.ftc.gov/news-events/press-releases/2013/12/android-flashlight-app-developer-settles-ftc-charges-it-deceived.

[7] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *7th USENIX Workshop on Hot Topics in Security*, 2012.

[8] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attentions, Comprehension, and Behavior. In *Symposium on Usable Privacy and Security*, 2012.

[9] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These aren't the droids you're looking for:" retrofitting Android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security*, 2011.

[10] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for Android apps. In *24th USENIX Security Symposium*, 2015.

[11] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *36th International Conference on Software Engineering*, 2014.

[12] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *21st USENIX Security Symposium*, 2012.

[13] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *NDSS Symposium*, 2014.

[14] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operating system framework for secure smartphones. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[15] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[16] K. K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking interaction-based declassification policies for android using symbolic execution. *CoRR*, abs/1504.03711, 2015.

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

[18] K. Onarlioglu, W. Robertson, and E. Kirda. Overhaul: Input-Driven Access Control for Better Privacy on Traditional Operating Systems. In *IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), 2016.

[19] Panda Security. Tapjacking - when the danger camouflages itself on google play. http://www.pandasecurity.com/mediacenter/tips/13973/, 2015.

[20] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *22nd USENIX Conference on Security*, 2013.

[21] Y. Qiu. Tapjacking: An untapped threat in android. http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/, 2012.

[22] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *ACM Conference on Computer and Communications Security*, 2014.

[23] F. Roesner and T. Kohno. Securing Embedded User Interfaces: Android and Beyond. In *22nd USENIX Security Symposium*, 2013.

[24] 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 on Security and Privacy*, 2012.

[25] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard. Covert communication in mobile applications (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[26] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS Trusted Window System. In *USENIX Security Symposium*, 2004.

[27] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: Virus-Safe Computing for Windows XP. *Communications of the ACM*, 49:83–88, Sept. 2006.

[28] H. J. Wang, A. Moshchuk, and A. Bush. Convergence of Desktop and Web Applications on a Multi-Service OS. In *USENIX Workshop on Hot Topics in Security*, 2009.

[29] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *22nd USENIX Conference on Security*, 2013.

[30] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM Conference on Computer and Communications Security*, 2014.

[31] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium*, 2015.

[32] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *37th International Conference on Software Engineering*, 2015.

[33] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *ACM Conference on Computer and Communications Security*, 2013.

[34] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *ACM Conference on Computer and Communications Security*, 2014.

[35] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.