AWARE: Preventing Abuse of Privacy-Sensitive Sensors via Operation Bindings

Giuseppe Petracca
Pennsylvania State University, US
gxp18@cse.psu.edu

Ahmad-Atamli Reineh
University of Oxford, UK
atamli@cs.ox.ac.uk

Yuqiong Sun
Symantec Research Labs, US
yuqiong_sun@symantec.com

Jens Grossklags
Technical University of Munich, DE
jens.grossklags@in.tum.de

Trent Jaeger
Pennsylvania State University, US
tjaeger@cse.psu.edu

Abstract

System designers have long struggled with the challenge of determining how to control when untrusted applications may perform operations using privacy-sensitive sensors securely and effectively. Current systems request that users authorize such operations once (i.e., on install or first use), but malicious applications may abuse such authorizations to collect data stealthily using such sensors. Proposed research methods enable systems to infer the operations associated with user input events, but malicious applications may still trick users into allowing unexpected, stealthy operations. To prevent users from being tricked, we propose to bind applications' operation requests to the associated user input events and how they were obtained explicitly, enabling users to authorize operations on privacy-sensitive sensors unambiguously and reuse such authorizations. To demonstrate this approach, we implement the AWARE authorization framework for Android, extending the Android Middleware to control access to privacy-sensitive sensors. We evaluate the effectiveness of AWARE in: (1) a laboratory-based user study, finding that at most 7% of the users were tricked by examples of four types of attacks when using AWARE, instead of 85% on average for prior approaches; (2) a field study, showing that the user authorization effort increases by only 2.28 decisions on average per application; (3) a compatibility study with 1,000 of the most-downloaded Android applications, demonstrating that such applications can operate effectively under AWARE.

1 Introduction

Contemporary desktop, web, and mobile operating systems are continually increasing support for applications to allow access to privacy-sensitive sensors, such as cameras, microphones, and touch-screens to provide new useful features. For example, insurance and banking applications now utilize mobile platforms' cameras to collect sensitive information to expedite claim processing[1] and check depositing[2], respectively. Several desktop and mobile applications provide screen sharing[3] and screen capturing features for remote collaboration or remote control of desktop and mobile platforms. Also, web search engines now embed buttons to call the businesses linked to the results directly.

Unfortunately, once an application is granted access to perform such sensitive operations (e.g., on installation or first use), the application may use the operation at will, opening opportunities for abuse. Indeed, cybercriminals have built malware applications available online for purchase, called *Remote Access Trojans* (RATs), that *abuse authorized access* to such sensors to exfiltrate audio, video, screen content, and more, from desktop and mobile platforms. Since 75% of operations requiring permissions are performed when the screen is off, or applications are running in the background as services [54], these attacks often go unnoticed by users. Two popular RAT applications, widely discussed in security blogs and by anti-virus companies, are *Dendroid* [1] and *Krysanec* [19]. In the "Dendroid case", the Federal Bureau of Investigations and the Department of Homeland Security performed an investigation spanning several years in collaboration with law enforcement agencies in over 20 countries. The cybercriminal who pleaded guilty for spreading the malware to over 70,000 platforms worldwide was convicted of 10 years in prison and a $250,000 fine [16, 18]. Several other cases of abuse have been reported ever since. Some cases leading to legal actions, including the case of the NBA Golden State Warriors' free application that covertly turns on smartphones' microphones to listen to and record conversations [17], school laptops that were found to use their cameras to spy on students to whom they were given [13], and others [14, 15].

[1] Speed up your car insurance claim. www.esurance.com
[2] PNC Mobile Banking. www.pnc.com
[3] Remote Screen Sharing for Android Platforms. www.bomgar.com

Researchers have also designed mobile RAT applications to demonstrate limitations of access control models adopted by contemporary operating systems when mediating access to privacy-sensitive sensors. For instance, *PlaceRaider* [51] uses the camera and built-in sensors to construct three-dimensional models of indoor environments. *Soundcomber* [46] exfiltrates sensitive data, such as credit card and PIN numbers, from both tones and speech-based interaction with phone menu systems. Even the Meterpreter Metasploit exploit, enables microphone recording remotely on computers running Ubuntu[4].

To address these threats, researchers have proposed methods that enable the system to infer which operation requests are associated with which user input events. Input-Driven [33] access control authorizes the operation request that immediately follows a user input event, but a malicious application may steal a user input event targeted at another application by submitting its request first. User-Driven [39, 41] access control requires that applications use system-defined gadgets associated with particular operations to enable the system to infer operations for user input events unambiguously, but does not enable a user to verify the operation that she has requested by providing input. We describe four types of attacks that are still possible when using these proposed defenses.

In this work, we propose the AWare authorization framework to prevent abuse of privacy-sensitive sensors by malicious applications. Our goal is to enable users to verify that applications' operation requests correspond to the users' expectations explicitly, which is a desired objective of access control research [24, 28]. To achieve our objective, AWare binds each operation request to a user input event and obtains explicit authorization for the combination of operation request, user input event, and the user interface configuration used to elicit the event, which we call an *operation binding*. The user's authorization decision for an operation binding is recorded and may be reused as long as the application always uses the same operation binding to request the same operation. In this paper, we study how to leverage various features of the user interface to monitor how user input events are elicited, and reduce the attack options available to adversaries significantly. Examples of features include the widget selected, the window configuration containing the widget, and the transitions among windows owned by the application presenting the widget. In addition, AWare is designed to be completely transparent to applications, so applications require no modification run under AWare control, encouraging adoption for contemporary operating systems.

We implement a prototype of the AWare authorization framework by modifying a recent version of the Android operating system and found, through a study of 1,000 of the most-downloaded Android applications, that such applications can operate effectively under AWare while incurring less than 4% performance overhead on microbenchmarks. We conducted a laboratory-based user study involving 90 human subjects to evaluate the effectiveness of AWare against attacks from RAT applications. We found that at most 7% of the user study participants were tricked by examples of four types of attacks when using AWare, while 85% of the participants were tricked when using alternative approaches on average. We also conducted a field-based user study involving 24 human subjects to measure the decision overhead imposed on users when using AWare in real-world scenarios. We found that the study participants only had to make 2.28 additional decisions on average per application for the entire study period.

In summary, the contributions of our research are:

- We identify four types of attacks that malicious applications may still use to obtain access to privacy-sensitive sensors despite proposed research defenses.
- We propose AWare, an authorization framework to prevent abuse of privacy-sensitive sensors by malicious applications. AWare binds application requests to the user interface configurations used to elicit user inputs for the requests in *operation bindings*. Users then authorize operation bindings, which may be reused as long as the operation is requested using the same operation binding.
- We implement AWare as an Android prototype and test its compatibility and performance for 1,000 of the most-downloaded Android applications. We also evaluate its effectiveness with a laboratory-based user study, and measure the decision overhead imposed on users with a field-based user study.

2 Background

Mobile platforms require user authorization for untrusted applications to perform sensitive operations. Mobile platforms only request such user authorizations once, either at application installation time or at first use of the operation [2, 3] to avoid annoying users.

The problem is that malicious applications may abuse such blanket user authorizations to perform authorized, sensitive operations stealthily, without users' knowledge and at times that the users may not approve. Operations that utilize sensors that enable recording of private user actions, such as the microphone, camera, screen, etc., are particularly vulnerable to such abuse. Research studies have shown that such attacks are feasible in real systems, such as in commodity web browsers and mobile apps [21, 29, 37]. These studies report that more than 78%

[4] null-byte.wonderhowto.com

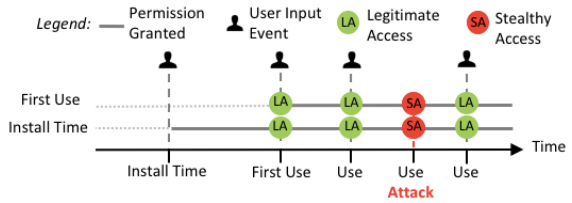

Figure 1: In mobile platforms, once the system authorizes an application to perform a operation, the application may perform that operation at any time, enabling adversaries to stealthily access privacy-sensitive sensors, e.g., record speech using the microphone, at any time.

of users could be potentially subject to such attacks. Furthermore, security companies, such as Check Point, have reported several malware apps that performs stealthy and fraudulent auto-clicking [4], such as `Judy`, `FalseGuide`, and `Skinner` that reached between 4.5 million and 18.5 million devices worldwide. Figure 1 shows that once an application is granted permission to perform an operation using a privacy-sensitive sensor, such as recording via the microphone, that application may perform that operation at any time, even without user consent. This shortcoming enables adversaries to compromise user privacy, e.g., record the user's voice and the surrounding environment, without the user being aware. Research studies have already shown that users have a limited understanding of security and privacy risks deriving from installing applications and granting them permissions [10].

Research [45, 51, 52] and real-world [1, 19] developers have produced exploits, called *Remote Access Trojans* (RATs), that abuse authorized operations to extract audio, video, screen content, etc., from personal devices while running in the background to evade detection by users. Instances of permission abuse have been reported in several popular mobile applications such as `Shazam`, `TuneIn Radio`, and `WhatsApp` [48].

Researchers have proposed defenses to prevent stealthy misuse of operations that use privacy-sensitive sensors [33, 39, 41]. Figure 1 also provides the insight behind these defenses: legitimate use of these sensors must be accompanied by a user input event to grant approval for all operations targeting privacy-sensitive sensors. First, Input-Driven Access Control [33] (IDAC) requires every application request for a sensor operation to follow a user input event within a restricted time window. Thus, IDAC would deny the stealthy accesses shown in Figure 1 because there is no accompanying user input event. Second, User-Driven Access Control [39, 41] (UDAC) further restricts applications to use trusted access control gadgets provided by the operating system, where each access control gadget is associated with a specific operation for such sensors. Thus, UDAC requires a user input event and limits the requesting application only to perform the operation associated with the gadget (i.e., widget) by the system.

3 Problem Definition

Although researchers have raised the bar for stealthy misuse of sensors, malicious applications may still leverage the user as the weak link to circumvent these protection mechanisms. Previous research [6, 12, 30] and our user study (see Section 8.1.1) show that users frequently fail to identify the application requesting sensor access, the user input widget eliciting the request, and/or the actual operation being requested by an application. Such errors may be caused by several factors, such as users failing to detect phishing [12], failing to recognize subtle changes in the interface [20], and/or failing to understand the operations granted by a particular interface [38]. In this section, we examine attacks that are still possible given proposed research solutions, and what aspects of proposed solutions remain as limitations.

3.1 User Interface Attacks

In this research, we identify four types of attacks that malicious applications may use to circumvent the protection mechanisms proposed in prior work [33, 39, 41].

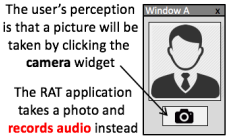

Figure 2: The user's perception of the operation that is going to be performed differs from the actual operation requested by the application, which abuses a previous granted permission.

Operation Switching: A malicious application may try to trick a user into authorizing an unintended operation by changing the mapping between a widget and the associated operation, as shown in Figure 2. This type of attack is possible in IDAC because the relationship between a user input event and the operation that will be authorized as a result of that event is implicit. Indeed, any application can request any operation for which they have been authorized previously (e.g., by first use) and will be approved if it is the first request received after the event. UDAC [39, 41] avoids this type of attack by design by having the system define a mapping between widgets (gadgets) and operations, so the operation is determined precisely by the widget. Any solution we devise must prevent this kind of attack as well.

Figure 3: A photo capturing application presents a video camera widget, instead of a camera widget, to trick the user into also granting access to the microphone. The windowing display context surrounding the widget shows a camera preview for photo capturing.

Bait-and-Context-Switch: A malicious application may try to trick the user to authorize an unintended operation by presenting a widget in a misleading display context, as shown in Figure 3. In this case, the win-

dowing context surrounding the widget indicates one action (e.g., taking a picture) when the widget presented requests access to a different operation (e.g., taking a video). This type of attack is possible because users engaged in interface-intensive tasks may focus on the context rather than the widget and infer the wrong widget is present, authorizing the wrong operation. Neither IDAC [33] nor UDAC [39, 41] detect the attack shown. Although UDAC [39] checks some properties of the display context[5], plenty of flexibility remains for an adversary to craft attacks since applications may choose the layout around which the widget is displayed.

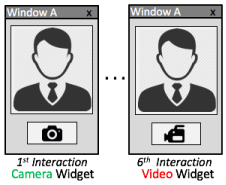

Figure 4: A malicious application keeps the windowing display context but switches the widget to trick users who have made several similar selections to grant the malicious application also access to the microphone mistakenly.

Bait-and-Widget-Switch: A malicious application may present the same widget for the same operation to the user several times in succession, but then substitute another widget for another operation, hoping that the user will not notice the widget change. An example of this attack is shown in Figure 4. Again, this type of attack is possible because users engaged in interface-intensive tasks may be distracted, thus, not notice changes in the widget. Again, UDAC methods to detect deceptive interfaces [39] are not restrictive enough to prevent this attack in general. For example, one UDAC check restricts the gadget's location for the user input event, but this does not preclude using different gadgets at the same location.

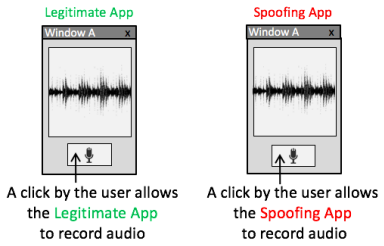

Figure 5: The user may mistakenly authorize access to the microphone to a RAT application spoofing the graphical aspect of a well-known legitimate application.

Application Spoofing: A malicious application replicates the look-and-feel of another application's interface and replaces the foreground activity of that application with one of its own to gain access to a sensor as shown in Figure 5, similar to a phishing attack. For example, when the benign application running in the foreground elicits a user input event, the malicious application may also try to elicit a user input event using its own activity window by replacing the benign application currently in the foreground. If the user supplies an input to the masquerading application's widget, then the masquerading application can perform any operation for which it is authorized (e.g., from first use or its manifest). While researchers have explored methods to increase the user's ability to detect the foreground application [6], mistakes are still possible. Indeed, prior studies have reported that few users notice the presence of security indicators, such as the browser lock icon [9, 53], and that even participants whose assets are at risk fail to react as recommended when security indicators are absent [44]. Since IDAC and UDAC [33, 39, 41] both treat user input as authorization, both will be prone to this attack[6].

3.2 Limitations of Current Defenses

The main challenge is determining when users allow applications to use particular privacy-sensitive sensors without creating too much burden on users. As a result, current mobile platforms only request user authorization once (e.g., on first use or installation), and proposed research solutions aim to infer whether users authorize access to particular sensors from user actions implicitly. However, inferring user intentions implicitly creates a semantic gap between what the system thinks the user intended and what the user actually intended.

Traditionally, access control determines whether subjects (e.g., users and applications) can perform operations (e.g., read and write) on resources (e.g., sensors). Proposed approaches extend traditional access control with additional requirements, such as the presence of a user input event [33, 41] or properties of the user interface [39]. However, some requirements may be difficult to verify, particularly for users, as described above, so these proposed approaches still grant adversaries significant flexibility to launch attacks. Proposed approaches still demand users to keep track of which application is in control, the operations associated with widgets, which widget is being displayed, and whether the widget or application changes.

Finally, application compatibility is a critical factor in adopting the proposed approaches. The UDAC solutions [39, 41] require developers to modify their applications to employ system-defined gadgets. It is hard to motivate an entire development community to make even small modifications to their applications, so solutions that do not require application modifications would be preferred, if secure enough.

4 Security Model

Trust Model - We assume that applications are isolated from each other either using separate processes, the same-origin policy [42], or sandboxing [7, 36], and have no direct access to privacy-sensitive sensors by default due to the use of Mandatory Access Control [49, 50].

[5]UDAC Audacious [39] checks that the user interface presented does not have misleading text, that the background and text preserve the contrast, and that the gadget is not covered by other user interface elements.

[6]UDAC authors [41] did acknowledge this attack, and indicated that solutions to such problems are orthogonal.

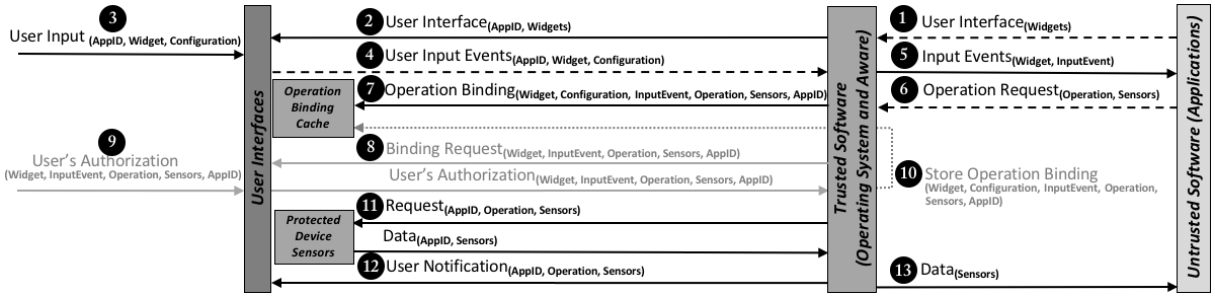

Figure 6: Overview of the AWARE authorization framework. The three dashed lines highlight the parts of information used by AWARE to generate an *operation binding*. The gray arrows represent *one-time* steps required to obtain an explicit authorization from the user for the creation of a new *operation binding*, which are not required when the *operation binding* has been explicitly authorized by the user in previous interactions.

We assume the presence of a *trusted path* for users to receive unforgeable communications from the system and provide unforgeable user input events to the system. We assume that trusted paths are protected by mandatory access control [49, 50] as well, which ensures that only trusted software can receive input events from trusted system input devices to guarantee the authenticity (i.e., prevent forgery) of user input events.

Trusted path communication from the system to the user uses a *trusted display area* of the user interface, which we assume is available to display messages for users and applications do not have any control of the content displayed in this area; thus they cannot interfere with system communications to or overlay content over the trusted display area.

These assumptions are in line with existing research that addresses the problem of designing and building trusted paths and trusted user interfaces for browsers [55], X window systems [47, 56], and mobile operating systems [26, 27]. The design of our prototype leverages mechanisms provided by the Android operating system satisfying the above assumptions, as better described in Section 7.

Threat Model - We assume that applications may choose to present any user interface to users to obtain user input events, and applications may choose any operation requests upon any sensors. Applications may deploy user interfaces that are purposely designed to be similar to that of another application, and replay its user interface when another application is running to trick the user into interacting with such interface to "steal" such user input event. Applications may also submit any operation request at any time when that application is running, even without a corresponding user input event. Applications may change the operation requests they make in response to user input events.

5 Research Overview

Our objective is to develop an authorization mechanism that eliminates ambiguity between user input events and the operations granted to untrusted applications via those events, while satisfying the following security, usability, and compatibility properties:

User Initiation Every operation on privacy-sensitive sensors must be initiated by an authentic user input event.

User Authorization Each operation on privacy-sensitive sensors requested by each application must be authorized by the user explicitly prior to that operation being performed.

Limited User Effort Ideally, only one explicit user authorization request should be necessary for any benign application to perform an operation targeting privacy-sensitive sensors while satisfying the properties above.

Application Compatibility No application code should require modification to satisfy the properties above.

We aim to control access to privacy-sensitive sensors that operate in discrete time intervals initiated by the user, such as the cameras, microphone, and screen buffers. We believe the control of access to continuous sensors, such as GPS, gyroscope, and accelerometer, requires a different approach [34], but we leave this investigation as future work.

To achieve these objectives, we design the AWARE authorization framework. The main insight of the AWARE design is to extend the notion of an authorization tuple (i.e., subject, resource, operation) used to determine whether to authorize an application's operation request to include the user interface configuration used to elicit the user input event. We call these extended authorization tuples *operation bindings*, and users explicitly authorize operation bindings before applications are allowed to access sensors. An operation binding may reused to authorize subsequent operations as long the application uses the same user interface configuration to elicit input events to request the same operation.

Approach Overview. Figure 6 summarizes the steps taken by the AWARE to authorize applications' operation requests targeting privacy-sensitive sensors.

In a typical workflow, an application starts by specifying a set of user interface configuration, such as widgets and window features, to the trusted software (step ❶) in charge of rendering such widgets with windows to elicit user input (step ❷). An authentic user interaction with the application's widgets in a user interface configuration generates user input events (step ❸), which are captured

by the trusted software (step 4) together with the current user interface configuration (e.g., enclosing window, window features, ancestors windows, etc.) and forwarded to the application (step 5). Based on the user input events, the application may generate a request for a particular operation targeting one or more privacy-sensitive sensors, which is captured by the trusted software (step 6).

At this stage, the AWare authorization framework (part of the trusted software layer) has full visibility of: (1) the application's identity; (2) the application's user interface widget; (3) the authentic user input event associated with that widget; (4) the user interface configuration within which the widget is presented to the user; (5) the application's operation request; and (6) the target set of privacy-sensitive sensors for such an operation. Thus, the AWare authorization framework can *bind* these pieces of information together, creating an *operation binding*.

Next, the AWare authorization framework checks whether such an operation binding has already been authorized by the user (step 7). If not, AWare presents a request for authorization of the operation binding to the user (Section 7), called the *binding request* (step 8). Upon receiving a binding request, the user can *explicitly* authorize the use of the set of privacy-sensitive sensors by the requesting application for the identified operation binding (step 9). Upon the user's authorization, the operation binding is then cached (Section 6.5) for reuse in authorizing future requests using the same operation binding automatically (step 10).

After the operation authorization, the trusted software controlling the set of privacy-sensitive sensors starts the data collection (step 11), while the user is *explicitly* notified about the ongoing operation via an on-screen notifications in a trusted display area (step 12). Finally, the collected data is delivered to the requesting application for data processing (step 13).

The sequence of events in Figure 6 shows that AWare relies on a one-time, explicit user authorization that binds the user input event, the application identity, the widget, the widget's user interface configuration, the operation, and the set of target sensors; then, it reuses this authorization for future operation requests.

6 AWare Design

6.1 Operation Bindings

As described above, AWare performs authorization using a concept called the operation binding.

Definition 1: An *operation binding* is a tuple $b = (app, S, op, e, w, c)$, where: (1) app is the application associated with both the user interface widget and the operation request; (2) S is the set of sensors (i.e., resources) targeted by the request; (3) op is the operation being requested on the sensors; (4) e is the user input event; (5) w is a user interface widget associated with the user input event; (6) c is the user interface configuration containing the widget.

The user interface configuration describes the structure of the user interface when a user input event is produced, which includdes both features of the window in which the widget is displayed and application's activity window call graph, which relates the windows used by the application. We define these two aspects of the configuration precisely and describe their use to prevent attacks in Sections 6.3 and 6.4.

The first part of an operation binding corresponds to the traditional authorization tuple of (subject, object, operation). An operating binding links a traditional operation tuple with a user input event and how it was obtained in terms of the rest of the operation binding tuple (event e, widget w, configuration c). AWare's authorization process enables users to authorize operation requests for the authorization tuple part of the operation binding (app, S, op) associated with a particular way the user approved the operation from the rest of the operation binding (e, w, c). AWare reuses that authorization to permit subsequent operation requests by the same application when user input events are obtained in the same manner.

A user's authorization of an operation binding implies that *the application will be allowed to perform the requested operation on the set of sensors whenever the user produces the same input event using the same widget within the same user interface configuration*.

We explain the reasoning behind the operation binding design by describing how AWare prevents the attacks described in Section 3.1 in the following subsections.

6.2 Preventing Operation Switching

AWare prevents operation switching attacks by producing an operation binding that associates a user input event and widget with an application's operation request.

Upon a user input event e, AWare collects the widget w, the user interface configuration c in which it is presented, and the application associated with the user interface app. With this partial operation binding, AWare awaits an operation request. Should the application make an operation request within a limited time window [33], AWare collects the application app, operation sensors S, and operation requested op, the traditional authorization tuple, to complete the operation binding for this operation request.

The constructed operation binding must be explicitly authorized by the user. To do so, AWare constructs a *binding request* that it presents to the user on the platform's screen. The binding request clearly specifies: (1) the identity of the requesting application; (2) the set of sensors targeted by the operation request; (3) the type of operation requested by the application; and (4) the widget receiving the user input event action.

This approach ensures that the user authorizes the combination of these four components enabling the user to verify the association between the operation being authorized and the widget used to initiate that operation. Also, each operation binding is associated with the specific user interface configuration for the widget used to activate the operation. Although, this information is not presented to the user, it is stored for AWare to compare to future operation requests to prevent more complex attacks, as described below.

This prevents the operation switching attack on IDAC [33], where another operation may be authorized by a user input event. AWare creates a binding between a widget and operation as UDAC [39, 41] does, but unlike UDAC AWare creates these bindings dynamically. Applications are allowed to choose the widgets to associate with particular operations. In addition, AWare informs the user explicitly of the operation to be authorized for that widget, whereas UDAC demands that the user learn the bindings between widgets and operations correctly. The cost is that AWare requires an explicit user authorization on the first use of the widget for an operation request, whereas UDAC does not. However, as long as this application makes the same operation requests for user input events associated with the same widget, AWare will authorize those requests without further user effort.

6.3 Preventing Bait-and-Switch

Applications control their user interfaces, so they may exploit this freedom to perform bait-and-switch attacks by either presenting the widget in a misleading window (Bait-And-Context-Switch) or by replacing the widget associated with a particular window (Bait-And-Widget-Switch). Research studies have shown that such attacks are feasible in real systems and that the damage may be significant in practice [21, 29, 37]. To prevent such attacks, AWare binds the operation request with the user interface configuration used to display the widget, in addition to the widget and user input event.

One aspect of the user interface configuration of the operation binding describes features of the window enclosing the widget.

Definition 2: A *display context* is a set of structural features of the most enclosing activity window *a_w* containing the widget *w*.

Structural features describe how the window is presented, excepting the content (e.g., text and figures inside web pages), which includes the position, background, borders, title information, and widgets' position within the window. The set of structural features used by AWare are listed in Table 5. AWare identifies *a_w* as a new activity window should any of these structural features change.

The hypothesis is that the look-and-feel of an application window defined by its structural features should be

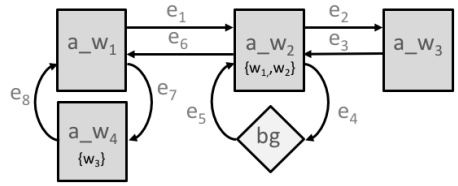

Figure 7: Activity window call graphs are created at runtime for the activity windows that produce authorized operations. (bg) is the background activity.

constant, while the content may change. Our examination of Android applications shows that the same windows retain the same look-and-feel consistently, but not perfectly. For example, the exact location of the window may vary slightly, so we consider allowing modest ranges for some feature values. We further discuss the authentication of display context in Section 7.

This approach prevents Bait-and-Widget-Switch attacks because clearly an instance of the same window (i.e., display context) with a different widget will not match the previous operation binding. Similarly, for Bait-and-Context-Switch attacks, the same widget presented in a different window (i.e., display context) will not match the previous operation binding, therefore a new operation binding request will be prompted to the user.

Once the widget and the display context are bound together and kept fixed, the adversary is left only with the content (e.g., text and figures inside a web page) as possible misleading tool. However, since the display context also measures the window's UI elements and their positions, little space is left to the adversary for attacks.

Therefore, such an approach prevents bait-and-switch attacks possible in both IDAC [33] and UDAC [39, 41], where users must continuously check for subtle changes to the widgets or their display contexts rendered on the platform's screen.

6.4 Preventing Application Spoofing

To launch such an attack an application must succeed in replacing the foreground activity window of one application with its own activity window and adopt the same look-and-feel of the replaced application.

We can prevent applications from presenting their activity windows arbitrarily by enforcing the application's authorized activity window call sequences.

Definition 3: An *activity window call graph* $G := (N, E)$ is a graph, where each node in N represents an activity window and each edge in E represents an inter-activity window transition enabled either via user input events (i.e., click of a button) or system events (i.e., incoming phone call).

An activity window call graph records the relationships among the windows used by an application. An example of an activity window call graph is shown in Figure 7, where events may cause transitions between windows

a_w_1 and a_w_4 and the application may enter the background only from the activity window a_w_2. Note that an application's activity window call graph can be built while the application runs, as the user authorizes operation bindings.

If the malicious application has not used this spoofing window previously, then a *binding request* will be created for the user, which then shows the identity of the application. Thus, the user could prevent the malicious application from ever performing this operation in any user interface configuration. IDAC [33] and UDAC [39, 41] do not explicitly force the user to check the application that will be authorized, although UDAC identified the need for such a mechanism [41].

On the other hand, a malicious application may try to hijack a foreground activity window of another application for a window that has been authorized by the user previously. However, if the malicious application's window is not authorized to transition from the background (e.g., only the activity window a_w_2 is authorized in Figure 7), then the transition will not match the activity call graph. In this case, a new *binding request* will be made to the user, which will clearly identify the (malicious) application. We discuss the authentication of the app identity in Section 7. Both IDAC and UDAC allow such hijacking and rely on the user to detect these subtle attacks.

A malicious application may try to circumvent the activity call graph checking by creating a more fully connected graph that allows more attack paths. However, such an activity window call graph will require more user authorizations, which may dissuade the user from that application. Furthermore, intrusion analysis may leverage such activity window call graphs to detect possible attacks.

6.5 Reusing Operation Bindings

Authorized *operation bindings* are cached to minimize the user's effort in making explicit authorizations of *binding requests* to improve usability. Thus, AWare uses a caching mechanism to require an explicit user's authorization only the first time an *operation binding* is identified, similarly to the first-use permission mechanism. We hypothesize that in most benign scenarios an authentic user interaction with a specific application's widget is going to generate a request for the same operation for the same set of privacy-sensitive sensors each time. Hence, the previous explicit authorization can be reused securely as *implicit* authorization, as long as the integrity of the *operation binding* is guaranteed. In Section 8.1.2, we show that such an approach does not prohibitively increase the number of access control decisions that users need to make thus avoiding *decision fatigue* [11].

However, we must ensure that *operation bindings* do not become stale. For example, if the application changes the way it elicits an operation, we should not allow the application to reuse old methods to elicit that same operation. Thus, we require that an *operation binding* must be removed from the cache whenever a new operation binding is created for the same application that partially matches the existing binding, except for the application field. For example, this prevents an operation from being authorized in multiple ways, a widget from being used for multiple operations or in multiple configurations, etc.

6.6 Supporting Existing Applications

As an alternative to previously proposed approaches [39, 41], AWare is completely transparent to, and backward compatible with, existing applications. In fact, AWare does not require any new external libraries, application code annotation or rewriting, which would require significant development effort/burden and impede backward compatibility for existing applications.

AWare can be integrated with existing off-the-shelf operating systems, as we show with our AWare prototype discussed in Section 7. AWare only requires the integration of three software components at the middleware layer. AWare's components dynamically monitor the creation of operation bindings and provide visual output to the user to enable authorization of operations on privacy-sensitive sensors. The integration with existing off-the-shelf operating systems facilitates adoption and deployability.

We discuss how AWare addresses special cases of applications accessing privacy-sensitive sensors via alternative methods, such as via background processes and remote commands, in Appendix A .

7 AWare Implementation

We implemented an AWare prototype by modifying a recent release of the Android operating system (version 6.0.1_r5) available via the Android Open Source Project (AOSP)[7]. The AWare prototype is open-sourced on github.com[8]. Its footprint is about 500 SLOC in C, 800 SLOC in C++ and 600 SLOC in Java. We tested the AWare prototype on Nexus 5 and Nexus 5X smartphones.

In the following paragraphs, we describe how we implemented the components required for AWare authorization mechanism[9].

Application Identity: To prove an app's identity in *binding requests*, AWare applies two methods. First, AWare uses the checksum of the app's binary signed with the developer's private key and verifiable with the developer's public key [40], similarly to proposals in related work [6]. In addition, AWare detects spoofing of apps' names or identity marks by using the Comparison Algo-

[7] https://source.android.com

[8] https://github.com/gxp18/AWare

[9] For brevity, in this and the following sections, we use the abbreviation *app* to refer to an application.

rithm for Navigating Digital Image Databases (CANDID) [25]. This comparison ensures that malicious apps do not use the same name or identity mark of other official apps. AWARE collects the developers' signatures and the apps identity marks (names and logos) from the Google Play store.

Widget and Display Context Authentication: AWARE identifies application-defined widgets and display contexts at runtime before rendering the app's user interface to the user on the platform's screen. AWARE uses the widget and window objects created in memory by the Window Manager, before rendering them on the platform's screen, to collect their graphical features reliably. A secure operating systems must prevent apps from being able to directly write into the frame buffers read by the hardware composer, which composes and renders graphic user interfaces on the platform screen. Modern operating systems, such as the Android OS, leverage mandatory access control mechanisms (i.e., SELinux rules) to guarantee that security sensitive device files are only accessible by trusted software, such as the Window Manager. Therefore, as shown in Figure 6, although apps can specify the graphic components that should compose their user interfaces, only the Window Manager, a trusted Android service, can directly write into the screen buffers subsequently processed by the hardware composer. Thus, the Window Manager is the man-in-the-middle and controls what apps are rendering on screen via their user interfaces. In the Appendix, Tables 4 and Table 5 show comprehensive sets of widgets and windows' features used by AWARE to authenticate the widgets and their display contexts.

Activity Window Call Graph Construction: At runtime, AWARE detects inter-activity transitions necessary to construct the per-application activity window call graph by instrumenting the Android Activity Manager and Window Manager components. Also, AWARE captures user input events and system events by instrumenting the Android Input Manager and the Event Manager components. We discuss nested activity windows in Appendix C.

User Input Event Authentication: AWARE leverages SEAndroid [49] to ensure that processes running apps or as background services cannot directly read or write input events from input device files (i.e., `/dev/input/*`) corresponding to hardware interfaces attached to the mobile platform. Thus, only the Android Input Manager, a trusted system service, can read such files and forward input events to apps. Also, AWARE leverages the Android screen overlay mechanism to detect when apps or background services draw over the app currently in the foreground to prevent input hijacking and avoid processing of any user input event on overlaid GUI widgets. Thus, AWARE considers user input events for the identification of an operation binding only if the widget and the corresponding window

Figure 8: AWARE *Binding Request* prompted to the user on the mobile platform's screen at *Operation Binding* creation. The app's identity is proved by the name and the graphical mark. For better security, in mobile platforms equipped with a fingerprint scanner, AWARE recognizes the device owner's fingerprint as the only authorized input for creating a new *Operation Binding*.

are fully visible on the platform's screen foreground. To intercept user input events, we placed twelve hooks inside the stock Android Input Manager.

Operation Request Mediation: The Hardware Abstraction Layer (HAL) implements an interface that allows system services and privileged processes to access privacy-sensitive sensors indirectly via well-defined APIs exposed by the kernel. Further, SEAndroid [49] ensures that only system services can communicate with the HAL at runtime. Thus, apps must interact with such system services to request execution of specific operations targeting privacy-sensitive sensors. Thus, AWARE leverages the complete mediation guaranteed at the system services layer to identify operation requests generated by apps at runtime, using ten hooks inside the stock Android Audio System, Media Server, and Media Projection.

Operation Binding Management: The AWARE prototype implements the AWARE MONITOR to handle callbacks from the AWARE hooks inside the Input Manager and other system services. The AWARE MONITOR is notified of user input events and apps' requests to access privacy-sensitive sensors via a callback mechanism. Also, the AWARE MONITOR implements the logic for the *operation binding* creation and caching as well as the display of *binding requests* and alerts to the user. User approvals for *binding requests* are obtained by the AWARE MONITOR via authorization messages prompted to the user on the mobile platform's screen, as shown in Figure 8. To protect the integrity of the trusted path for binding requests, we prevent apps from creating windows that overlap the AWARE windows or modifying AWARE windows. To prevent overlapping, AWARE leverages the Android screen overlay protection mechanism. To prevent unauthorized modification, AWARE implements the Compartmented Mode Workstation model [8] by using isolated per-window processes forked from the Window Manager.

7.1 Control Points Available to the User

AWARE provides the users with control points during authorized use of privacy-sensitive sensors by apps. These control points allow the users to control the apps' use of sensors and correct possible mistakes made during the authorization process.

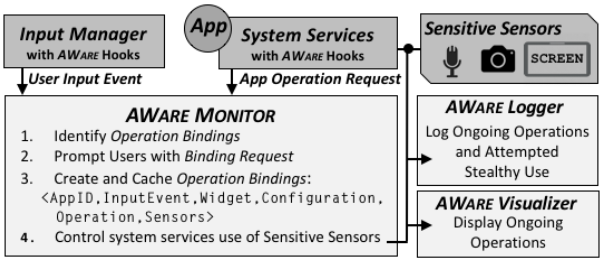

Figure 9: Architecture of the AWARE authorization framework.

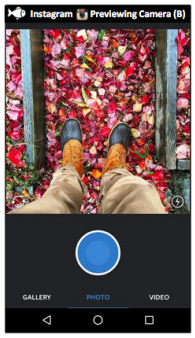

Figure 10: AWARE *security message* displayed on the mobile platform's status bar notifying the user that the Instagram application is previewing the back camera (B) for pictures. The security companion (e.g., a white fish) aids the user in verifying the authenticity of the authorization request. Each *security message* includes the app identifier (e.g., application name and identity mark) and a text message specifying the ongoing operation and the set of privacy-sensitive sensors being accessed.

Figure 9 shows an overview of the AWARE prototype components and how the control points are activated. The AWARE MONITOR is designed to activate the AWARE VISUALIZER and the AWARE LOGGER, upon the user authorization of an *operation binding*.

7.1.1 Visualizing Ongoing Operations

AWARE displays *security messages* on a reserved portion of the screen, drawable only by the Window Manager and not accessible by untrusted applications, to make ongoing use of privacy-sensitive sensors visible to users until they terminate. An example of *security message* is shown in Figure 10. A *security message* includes the app identifier (e.g., application name and identity mark) and a text message specifying the ongoing operation and the set of privacy-sensitive sensors being accessed. The use of *security messages* follows the principle of *what the user sees is what is happening* [23], in fact, *security messages* convey *ongoing operations* targeting privacy-sensitive sensors when authorized by the user.

AWARE leverages the Compartmented Mode Workstation principle [8] to ensure integrity and authenticity of *security messages*. Also, AWARE uses a security companion, a secret image chosen by the user, to aid users in verifying the authenticity of *security messages*. We modified the stock Android system user interface (SystemUI), by adding an image view and a text view on the Android status bar to display the AWARE *security messages* specifying the application IDs and the ongoing operations, whenever the AWARE MONITOR authorizes system services to operate on privacy-sensitive sensors on behalf of applications. Also, the AWARE prototype leverages the Android screen overlay mechanism to detect when applications or background services draw over the application currently in the foreground, to prevent GUI overlay.

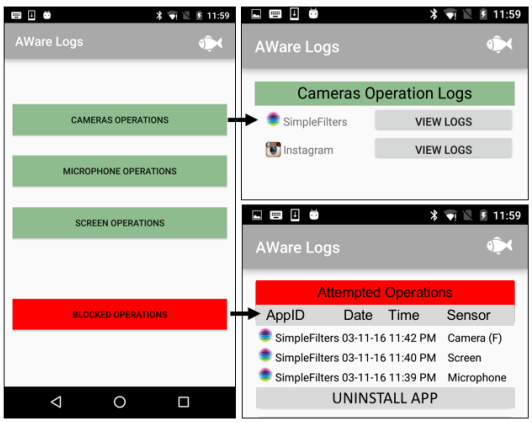

Figure 11: AWARE Users may leverage AWARE *logs* to take retrospective security actions. The figure at the top right shows the list of operations targeting the camera in authorized sessions. The figure at the bottom right summarizes attempted accesses to privacy-sensitive sensors by the SimpleFilters app, as examples of stealthy operations. The security companion chosen by the user (e.g., a white fish) aids the user in authenticating the logs.

Further, *security messages* are made visible to the user even if the application runs in full-screen mode. Reserving a small portion of the screen (5%) to convey a security message is a reasonable trade-off for preventing unwanted user distraction while delivering critical content in a timely and appropriate manner [32]. Our evaluation with existing full-screen applications (Section 8.1.2) reports that *security messages* do not impair the correct functioning of full-screen apps. A transparent background can also be used to reduce overlap with the foreground application's window. Lastly, the user can be given the option to explicitly replace the on-screen notification with a periodic distinctive sound or a hardware sensor-use indicator LED.

7.1.2 Logging Authorized Operations

AWARE produces real-time logs of any operation explicitly authorized by the user and of any attempted use of privacy-sensitive sensors from applications without a user-initiated input. AWARE makes attempted stealthy accesses, by installed applications, visible to users via full-screen alert messages and by producing a distinctive sound, or by enabling a hardware sensor-use indicator LED. AWARE then allows the user to either uninstall suspicious applications or to terminate ongoing suspicious operations. Logs are visible to users via a system application called AWARE LOGGER, which is accessible via the applications menu or by tapping on the AWARE security messages/alerts dis-

played on the mobile platform's screen. Each log entry reports information regarding the app ID, date, time, and the privacy-sensitive sensors target of the operation, as shown in Figure 11. Logs are not accessible to applications to preserve their integrity and avoid the creation of side channels.

8 AWARE Evaluation

We investigated the following research questions.

To what degree is the AWARE *operation binding concept assisting the users in avoiding attacks?* We performed a laboratory-based user study and found that the operation binding enforced by AWARE significantly raised the bar for malicious apps trying to trick the users in authorizing unintended operations, going from an average attack success rate of 85% down to 7%, on average, with AWARE.

What is the decision overhead imposed to users due to per-configuration access control? We performed a *field-based* user study and found that the number of decisions imposed to users by AWARE remains confined to less than four decisions per app, on average, for the study period.

How many existing apps malfunction due to the integration of AWARE*? How many operations from legitimate apps are incorrectly blocked by* AWARE *(i.e., false positives)?* We used a well-known compatibility test suite to evaluate the compatibility of AWARE with existing apps and found that, out of 1,000 apps analyzed, only five of them malfunctioned due to attempted operations that AWARE blocked as potentially malicious. However, these malfunctioning instances have been resolved by features developed in subsequent versions of the AWARE prototype.

What is the performance overhead imposed by AWARE *for the operation binding construction and enforcement?* We used a well-known software exerciser to measure the performance overhead imposed by AWARE. We found that AWARE introduced a negligible overhead on the order of microseconds that is likely to be not noticeable by users.

8.1 Preliminaries for the User Studies

We designed our user studies following suggested practices for human subject studies in security to avoid common pitfalls in conducting and writing about security and privacy human subject research [43]. Participants were informed that the study was about mobile systems security, with a focus on audio and video, and that the involved researchers study operating systems security. An Institutional Review Board (IRB) approval was obtained from our institution. We recruited user study participants via local mailing lists, Craigslist, and local groups on Facebook, and compensated them with a $10 gift card. We excluded friends and acquaintances from participating in the studies to avoid acquiescence bias. Participants were given the option to withdraw their consent to participate at any time after the purpose of the study was revealed. For all the experiments, we configured the test environment on Nexus 5X smartphones and used a background service, automatically relaunched at boot time, to log participants' responses to system messages/alerts and all user input actions taken by participants while interacting with the testing apps.

8.1.1 Laboratory-Based User Study

We performed a *laboratory-based* user study to evaluate the *effectiveness* of AWARE in supporting users in avoiding attacks by malicious apps and compared it with alternative approaches.

We divided the participants into six groups. Participants in Group1 interacted with a *stock* Android OS using install-time permissions. Participants in Group2 interacted with a *stock* Android OS using first-use permissions. Participants in Group3 interacted with a *modified* version of the Android OS implementing input-driven access control, which binds user input events to the operation requested by an app but does not prove the app's identity to the user. Participants in Group4 interacted with a *modified* version of the Android OS implementing the first-use permissions and a security indicator that informs the users about the origin of the app (i.e., developer ID [6]). Participants in Group5 interacted with a *modified* version of the Android OS implementing the use of access control gadgets [41] including basic user interface configuration checks (i.e., no misleading text, UI background and the text must preserve the contrast, no overlay of UI elements, and user events occur in the correct location at the correct time [39]) and timing checks for implicit authorizations. Lastly, participants in Group6 interacted with a *modified* version of the Android OS integrating the AWARE authorization framework.

Experimental Procedures: Before starting the experiment, *all participants were informed that attacks targeting sensitive audio and video data were possible during the interaction with apps involved in the experimental tasks*, but none of the participants were aware of the attack source. Further, the order of the experimental tasks was randomized to avoid ordering bias. All the instructions to perform the experimental tasks were provided to participants via a handout at the beginning of the user study. Participants were given the freedom to *ignore task steps if they were suspicious* about the resulting app activities.

We used two apps, a well-known voice note recording app called Google Keep, and a testing app (developed in our research laboratory) called SimpleFilters, which provides useful photo/video filtering functionality. However, SimpleFilters also attempts adversarial use of privacy-sensitive sensors, such as the microphone and the

Task Description (Randomized)	Attack Scenario	Authorization Requests (△ AWARE)	Attack Success Rate
TASK 1 : Take a picture with the smartphone's front camera by using the SimpleFilters app.	Operation Switching: The SimpleFilters app also starts recording audio via the smartphone's microphone instead of only taking a picture.	• Allow SimpleFilters to use the Front Camera and Microphone to Record Video when pressing 📷 ?	Group1 (Install-Time): 100% Group2 (First-Use): 93% Group3 (Input-Driven): 100% Group4 (Developer ID): 100% Group5 (AC Gadgets): 0% Group6 (AWARE): 0%
TASK 2 : Take a picture with the front camera by using the SimpleFilters app.	Bait-and-Context-Switch: We make the video camera widget appear in the photo capture window, with a camera preview, to trick the user into allowing SimpleFilters to record audio instead of just take a picture. □	• Allow SimpleFilters to use the Front Camera and Microphone to Record Video when pressing 📹 ?	Group1 (Install-Time): 87% Group2 (First-Use): 87% Group3 (Input-Driven): 93% Group4 (Developer ID): 87% Group5 (AC Gadgets): 87% Group6 (AWARE): 7%
TASK 3 : Take six consecutive pictures with the smartphone's front camera by using the SimpleFilters app.	Bait-and-Widget-Switch: Before the participants took the fifth picture, the SimpleFilters app replaced the camera widget with the video camera widget to enable video recording instead. The camera button was restored before the users took the sixth picture. □	• Allow SimpleFilters to use the Front Camera and Microphone to record Video when pressing 📹 ?	Group1 (Install-Time): 87% Group2 (First-Use): 87% Group3 (Input-Driven): 93% Group4 (Developer ID): 87% Group5 (AC Gadgets): 87% Group6 (AWARE): 7%
TASK 4 : Record a voice note using the Keep app.	Identity Spoofing: We let the participants select the Keep app from the app menu, however, we programmatically triggered the SimpleFilters app to hijack the on-screen activity and spoof the Keep app.	• Allow SimpleFilters to use the Microphone to Record Audio when pressing 🎙 ?	Group1 (Install-Time): 93% Group2 (First-Use): 93% Group3 (Input-Driven): 93% Group4 (Developer ID): 47% Group5 (AC Gadgets): 93% Group6 (AWARE): 0%

Table 1: Experimental tasks for the laboratory-based user study to evaluate the effectiveness of AWARE in preventing four types of user interface attacks. The authorization requests reported in the third column are due to the fact that AWARE requests a new explicit authorization whenever a widget is presented within a new configuration. △ Participants from Groups6 received additional authorization requests because the widgets were presented within new configurations automatically identified by AWARE. □ The camera preview showed a static picture to simulate a photo capture during video recording.

camera. We explicitly asked the participants to install such apps on the testing platforms[10].

Before starting the experiment tasks, we asked the participants to familiarize themselves with Google Keep, by recording a voice note, and with SimpleFilters, by taking a picture and recording a video with the smartphone's front camera. *During this phase, participants were presented with authorization requests at first use of any of the privacy-sensitive sensors.*

All the user study participants in Groups1-6 were asked to perform the four experimental tasks reported in Table 1. We designed such tasks to test the four types of attacks discussed in Section 3.1. During the experiment, the researchers recorded whether the participants commented noticing any suspicious activity in the apps' user interface, while a background service logged whether the designed attacks took place.

Experimental Results: 90 subjects participated and completed our experimental tasks. We randomly assigned 15 participants to each group. The last column of Table 1 summarizes the results for the four experimental tasks used in the laboratory-based user study. The third column of Table 1 reports additional authorization requests prompted only to subjects in Group6 using the AWARE system. Indeed, only AWARE is able to identify the change in configuration (e.g., widget in a different activity window, widget linked to a different operation or different privacy-sensitive sensor) under which the experimental applications are attempting access to the privacy-sensitive sensors (i.e, microphone and cameras).

[10] SimpleFilters is providing interesting features to convince the users to install it and grant the required permissions.

Overall, we found that all the operation binding components used by AWARE were useful in helping the users in avoiding the four types of attacks. Moreover, AWARE outperformed alternative approaches conspicuously, while each experimental task revealed interesting facts.

In particular, the analysis of the subjects' responses to **TASK 1** revealed that the operation performed by the app was not visible to users in the alternative approaches, thus, leading them into making mistakes. The only exceptions were the subjects from Group5 (AC Gadgets) because the SimpleFilters app was not in control of the requested operation due to the use of a system-defined access control gadget. Furthermore, all subjects from Group6 (AWARE) did not authorize SimpleFilters to access the microphone. Thus, the *binding request* clearly identifying the operation requested by the app aided them in avoiding to be tricked into granting an unintended operation.

The analysis of the subjects' responses to **TASK2** and **TASK3** revealed that the users were successfully tricked by either switching the user interface configuration within which a widget is presented, or by changing the widget presented within the same configuration, thus, leading them into making mistakes. Interestingly, there was no noticeable improvement for subjects in Group5 (AC Gadgets) where the system put in place some basic user interface configuration checks [39] for the presentation of the access control gadgets. The reason was that such basic checks were insufficient to identify the user interface modifications made by the malicious app when performing the attacks described in Table 1. Furthermore, one subject from Group6 (AWARE) had mistakenly authorized SimpleFilters to carry out an unintended operation

App Category	App Name	Explicit User Authorizations		Total Operation Authorizations
		First-Use	AWARE	Avg. (s.d.)
Audio Recording	WhatsApp	3	6 (±1)	1,217 (±187)
	Viber	1	1 (±1)	88 (±9)
	Messenger	3	7 (±2)	2,134 (±176)
Photo and Video Recording	Facebook	2	4 (±1)	3,864 (±223)
	SilentEye	2	5 (±1)	234 (±20)
	Fideo	2	4 (±1)	213 (±23)
Screenshot Capture	Ok Screenshot	1	2 (±1)	49 (±8)
	Screenshot Easy	1	2 (±1)	76 (±7)
	Screenshot Capture	1	2 (±1)	64 (±4)
Screen Recording	REC Screen Recorder	2	3 (±1)	41 (±8)
	AZ Screen Recorder	2	4 (±2)	49 (±7)
	Rec.	2	3 (±1)	66 (±4)
Full Screen Mode	Instagram	2	6 (±1)	3,412 (±182)
	Snapchat	2	6 (±1)	5,287 (±334)
	Skype	2	9 (±3)	468 (±62)
Remote Control	Prey Anti Theft	2	8 (±2)	47 (±5)
	Lost Android	2	6 (±1)	37 (±6)
	Avast Anti-Theft	2	4 (±1)	34 (±7)
Hands-Free Control	Google Voice Search	1	1 (±1)	1,245 (±122)
	HappyShutter	1	1 (±0)	3 (±1)
	SnapClap	1	1 (±0)	4 (±2)

Table 2: Applications tested during the field-based user study, selected among the most popular apps from the Google Play store. The last column reports the average and standard deviation for the total number of operation authorizations automatically granted by AWARE after the user's explicit authorizations. The values are rounded to ease the presentation.

even after receiving a *binding request* clearly identifying the operation. This event hints to the fact that users may still make mistakes even after they are given an explicit authorization request specifying the actual app-requested operation. However, users who make mistakes have still control points provided by AWARE via the *security messages* and *logs*, which allow addressing such mistakes by means of retrospective actions (Section 7.1).

Lastly, the analysis of the subjects' responses to **TASK 4** revealed that the real identity of the app performing the operation was not visible to users in the alternative approaches, thus, leading them into making mistakes. However, no subjects from Group6 (AWARE) authorized SimpleFilters to access the microphone. Therefore, the security message including the app's identity aided the user in identifying the attack.

8.1.2 Field-Based User Study

We performed a *field-based* user study to address the concern that AWARE may increase the decision overhead imposed on users as a result of finer-grained access control. We measured the number of explicit authorizations users had to make when interacting with AWARE under realistic and practical conditions. We also measured the total number of authorizations handled by AWARE via the operation binding cache mechanism that, transparently to users, granted previously authorized operations.

Experimental Procedures: Participants were asked to use, for a period of one week, a Nexus 5X smartphone running a *modified* version of the Android OS integrating the AWARE authorization framework. During this period, participants interacted with 21 popular apps (i.e., average number of apps users have installed on personal smartphones[11]) selected among the most popular apps with up to millions of downloads from the Google Play store. A description of the functionality provided by each app was given to participants. We then asked participants to explore each app and interact as they would normally do. Table 2 summarizes all the apps that were pre-installed on the smartphones for the field-based user study. The smartphones provided to participants were running a background service with a run-time log enabled, automatically restarted at boot time, to monitor the number of app activations, the number of widgets per app, and the number of decisions per app made by the users.

Experimental Results: 24 subjects participated and completed the field-based user study. Table 2 reports the average number of explicit authorizations performed by the participants when using AWARE, for each of the 21 apps used in the field-based user study. We compare them with the number of explicit authorizations that would be necessary if the first-use permission mechanism was used instead. The results show that 4 apps required the same number of explicit authorizations as for the first-use permission approach. For the remaining 17 apps, the number of decisions imposed to the users remains very modest. Over the 21 apps, an average of 2.28 additional explicit user authorizations are required per app.

Also, as expected, the number of explicit authorizations made by the users remained a constant factor compared to the number of operation authorization requests, automatically granted by AWARE (last column of Table 2), which instead grew linearly during the experiment period. Indeed, all the successive authorizations were automatically granted by AWARE.

8.2 Compatibility Analysis

We used the Compatibility Test Suite (CTS)[12], an automated testing tool, to evaluate the compatibility of AWARE with 1,000 existing apps selected from the Google Play store among the most downloaded apps[13].

The experiment took 13 hours and 28 minutes to complete, and AWARE passed 126,681 of the 126,686 executed tests. Two of the failed tests were minor compatibility issues due to attempted programmatic accesses to the platform's camera and microphone, respectively. The first failure was due to HappyShutter, an app that automatically takes pictures when the user smiles. The second failure was due to SnapClap, an app that automatically takes snapshots when the user claps. By default, AWARE blocks apps from programmatically accessing privacy-

[11] https://www.statista.com/chart/1435/top-10-countries-by-app-usage/

[12] https://source.android.com/compatibility/cts/

[13] The Absolute 1,000 Top Apps for Android. http://bestapps

sensitive sensors by intercepting API calls from running apps and verifying if the user has indeed initiated the operation. These checks provide a high level of protection. Thankfully, as described in Appendix A, less than 1% of the 1,000 analyzed apps require programmatic access to privacy-sensitive sensors. However, we enhanced the original AWare prototype to notify the user the first time that a programmatic access is attempted by an app. Such notification asks the user for an explicit authorization to grant the app *persistent* access to the privacy-sensitive sensor. The user is notified of the inherent high risk and is discouraged from granting such type of permission. We evaluated such feature in our field-based study as reported in Table 2. From our experiments, we found that only 1 of the 24 users granted persistent access to the front camera for the HappyShutter app, whereas, only 2 other users granted persistent access to the microphone for the SnapClap app.

The other two failures were due to remote access to the smartphone's camera attempted by two apps, namely Lockwatch and Prey Anti-Theft, which can capture pictures with the front camera when someone tries to unlock the smartphone's screen with a wrong passcode. However, as described in Appendix A, we anticipated this issue and suggested the extension of the mechanisms provided by AWare also to the remote app components that enable remote access. To validate the proposed extension, we have developed a proof-of-concept app that receives remote commands for the initiation of video recording via the mobile platform's back camera. We successfully tested it on a Nexus 5X smartphone running the Android OS integrating AWare.

Lastly, AWare caused another spurious false positive with the Viber app, which attempted access to the cameras and microphone at each system reboot. AWare, identified the access without a user input action and blocked the operation after displaying an onscreen alert and logging the attempted operation. After analyzing the Viber app, we noticed that the app was testing the sensors (e.g., cameras and microphone) at each reboot. However, preventing the Viber app from using the sensors for testing purposes did not cause subsequent video or voice calls to fail. Thus, we believe that blocking such attempts is the desired behavior to prevent stealthy operations targeting privacy-sensitive sensors.

8.3 Performance Measurements

We measured the overall system performance overhead introduced by AWare by using a macrobenchmark that exercises the same 1,000 apps selected from the Google Play store via the Android UI/Application Exerciser Monkey[14]. Although software exercisers only achieve a low code coverage, they can create events that target specific high-level operations and generate the same sequence of events for comparison among several testing platforms. Indeed, the Monkey was configured to exercise apps by generating the exact same sequence of events and target all operations on privacy-sensitive sensors on both the Nexus 5X and Nexus 5 smartphones when running both the stock Android OS and the modified version of Android with AWare enabled. We open-sourced the exerciser script for the macrobanchmark on github.com[15].

The experimental results reported in the first row of Table 3 show that the average recorded system-wide performance overhead is 0.33% when measuring the additional time required by AWare to handle the operation binding construction, authorization and caching.

We also performed a microbenchmark to measure the overhead introduced by AWare while specifically handling access requests for operations targeting privacy-sensitive sensors, such as the camera to take photos and videos, the microphone to record audio, and the screen to capture screenshots; and to measured the overhead introduced for the authentication of app-specific widgets and their display contexts. The overhead for operations targeting privacy-sensitive sensors was calculated by measuring the time interval from the time a user input action was detected to the time the corresponding app request was granted/denied by AWare. Instead, the overhead for the widgets' and display contexts' authentication was calculated by measuring the time interval from the time the app provided the user interface to the Window Manager to the time such interface was rendered on the platform's screen by AWare. Table 3 reports the average time and stadard deviation over 10,000 operation/rendering requests, and the recorded overhead introduced by AWare.

Our measurements show that AWare performs efficiently, with the highest overhead observed being below 4%, as shown in Table 3. Notice, the experiment artificially stressed each operation with unusual workloads, and the overhead for a single operation/rendering is on the order of microseconds. Thus, the overhead is likely not to be noticeable by users.

Lastly, we recorded the average cache size used by AWare to store authorized operation bindings and the activity window call graphs, which was around 3 megabytes. Overall, we did not observe a discernible performance drop compared to the stock Android OS.

9 Related Work

Security-Enhanced Android [49] and *Android Security Framework* [5] deploy restrictive security models based on the *Android Permission* mechanism. However, such models mainly operate at the kernel level, therefore, do

[14] https://developer.android.com/studio/test/monkey.html

[15] https://github.com/gxp18/AWare

	Stock Android		AWare		
	Nexus 5	Nexus 5X	Nexus 5	Nexus 5X	Average Overhead
System-Wide	32,983.38 ±103.76	31,873.71 ±217.82	33,001.32 ±109.79	31,981.02 ±207.81	0.33%
Front Camera	15.90±1.54	14.39±1.12	16.11±1.77	15.01±1.38	3.22%
Back Camera	16.08±1.32	15.68±1.87	16.44±1.06	16.37±1.91	3.13%
Microphone	12.36±2.01	11.86±1.99	12.65±2.15	12.32±1.85	3.01%
Screen	17.76±0.99	16.23±0.69	18.61±0.90	17.02±1.01	3.98%
Widget	22.12±0.35	21.66±0.54	24.61±0.32	23.45±0.12	2.79%

Table 3: AWare performance overhead in microseconds (μs). Numbers give mean values and corresponding standard deviations after 5 independent runs for the system-wide experiment and after 10,000 independent requests for the device-specific microbenchmark.

not have the necessary information regarding higher level events required to associate app requests to user input actions for operations targeting privacy-sensitive sensors.

Input-Driven Access Control (IDAC) [33] mediates access to privacy-sensitive sensors based on the temporal proximity of user interactions and applications' access requests. However, if another application's request occurs first after a user input event and within the given temporal threshold, then the user input is directly used to authorize the other applications request, no matter what operation the application is requesting.

In *What You See is What They Get* [23] the authors propose the concept of a *sensor-access widget*. This widget is integrated into the user interface within an applications display and provides a real-time representation of the personal data being collected by a particular sensor to allow the user to pay attention to the application's attempt to collect the data. Also, a widget is a control point through which the user can configure the sensor to grant or deny the application access. Such widgets implement a so-called *Show Widget and Allow After Input and Delay* (SWAAID) policy. According to such policy, any active user input, upon notification, is implicitly considered as an indication that the user is paying attention to the widget. Thus, after a waiting period, the application is directly authorized to access the sensor. However, the delay introduced for the waiting time (necessary to allow explicit denial) may cause issues for time-constrained applications and may frustrate users.

User-Driven Access Control (UDAC) [39, 41] proposes the use of *access control gadgets* to prevent malicious operations from applications trying to access privacy-sensitive sensors without a user-initiated input. However, access control gadgets define the start points for when permissions are granted but do no provide an end limit for the sensor's use or control points (Section 7.1) to the users. Moreover, each sensor's usage should be limited to the particular configuration within which it has been authorized by the user and should be terminated when the application tries to continue using the sensor in a different configuration.

Researchers have also explored a *trusted output* solution to provide the user with an on-screen security indicator to convey the application developer's identity for the application with which the user is interacting [6]. Such a solution aids the user in identifying applications developed by trusted sources (i.e., Google Inc.), but it does not provide the user with the actual application identity or information about *when* and *how* such an application uses privacy-sensitive sensors.

Lastly, researchers have proposed a new operating system abstraction called *object recognizer* for Augmented Reality (AR) applications [22]. A trusted object recognizer takes raw sensor data as input and only exposes higher-level objects, such as a skeleton of a face, to applications. Then, a fine-grained permission system, based on the visualization of sensitive data provided to AR applications, is used to request permission at the granularity of recognizer objects. However, the proposed approach applies only to AR applications which are a very small fraction of the applications available on the app market. Indeed, among the 1,000 applications used for our evaluation, fewer than 1% of them provide AR features. All the other applications require full access to the raw data in order to function properly.

10 Conclusion

To prevent abuse of privacy-sensitive sensors by untrusted applications, we propose that user authorizations for operations on such sensors must be explicitly bound to user input events and how those events are obtained from the user (e.g., widgets and user interface configuration), called *operation bindings*. We design an access control mechanism that constructs operation bindings authentically and gains user approval for the application to perform operations only under their authorized operation bindings. By reusing such authorizations, as long as the application always requests that operation using the same user input event obtained in the same way, the number of explicit user authorizations can be reduced substantially. To demonstrate the approach, we implemented the AWare framework for Android, an extension of the Android Middleware that controls access to privacy-sensitive sensors. We evaluated the effectiveness of AWare for eliminating ambiguity in a laboratory-based user study, finding that users avoided mistakenly authorizing unwanted operations 93% of the time on average, compared to 19% on average when using proposed research methods and only 9% on average when using first-use or install-time authorizations. We further studied the compatibility of AWare with 1,000 of the most-downloaded Android applications and demonstrated that such applications can operate effectively under AWare while incurring less than 4% performance overhead on microbenchmarks. Thus, AWare offers users an effective additional layer of defense against untrusted applications with potentially malicious

purposes, while keeping the explicit authorization overhead very modest in ordinary cases.

Acknowledgements

Thanks to our shepherd Matt Fredrikson and the anonymous reviewers. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors andshould not be interpreted as representing the official policies,either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not with standing any copyright notation here on. The research activities of Jens Grossklags are supported by the German Institute for Trust and Safety on the Internet (DIVSI).

References

[1] Dendroid malware can take over your camera, record audio, and sneak into Google play. 2014.

[2] Runtime and security model for web applications. 2015.

[3] App permissions explained-what are they, how do they work, and should you really care? 2016.

[4] The Judy Malware: Possibly the largest malware campaign found on Google Play. 2017.

[5] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th annual computer security applications conference*, pages 46–55. ACM, 2014.

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the Android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948, May 2015.

[7] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, volume 91. Seattle, WA, 2000.

[8] P. T. Cummings, D. Fullan, M. Goldstien, M. Gosse, J. Picciotto, J. L. Woodward, and J. Wynn. Compartmented model workstation: Results through prototyping. In *Security and Privacy, 1987 IEEE Symposium on*, pages 2–2. IEEE, 1987.

[9] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[10] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[12] A. P. Felt and D. Wagner. Phishing on mobile devices, 2011.

[13] https://en.wikipedia.org/. Robbins v. Lower merion school district federal class action lawsuit.

[14] https://www.ftc.gov. FTC letters warn companies of privacy risks in audio monitoring technology. 2016.

[15] http://www.nytimes.com. How spy tech firms let governments see everything on a smartphone. 2016.

[16] http://www.scmagazine.com. Fireeye intern pleads guilty in darkode case. 2015.

[17] http://www.sfgate.com. Lawsuit claims popular warriors app accesses phone's microphone to eavesdrop on you. 2016.

[18] http://www.tripwire.com. Fireeye intern pleads guilty to selling dendroid malware on darkode. 2014.

[19] http://www.welivesecurity.com. Krysanec trojan: Android backdoor lurking inside legitimate apps. 2014.

[20] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 22–22, Berkeley, CA, USA, 2012. USENIX Association.

[21] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, pages 413–428, 2012.

[22] S. Jana, D. Molnar, A. Moshchuk, A. M. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security*, pages 415–430, 2013.

[23] S. S. Jon Howell. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy*. IEEE, May 2010.

[24] J. Jung, S. Han, and D. Wetherall. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 45–50. ACM, 2012.

[25] P. M. Kelly, T. M. Cannon, and D. R. Hush. Query by image example: the comparison algorithm for navigating digital image databases (candid) approach. In *IS&T/SPIE's Symposium on Electronic Imaging: Science & Technology*, pages 238–248. International Society for Optics and Photonics, 1995.

[26] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[27] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[28] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012.

[29] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security*, pages 227–243. Springer, 2012.

[30] L. Malisa, K. Kostiainen, and S. Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. *IACR Cryptology ePrint Archive*, 2015:709, 2015.

[31] B. McCarty. *SElinux: NSA's open source security enhanced linux*. O'Reilly Media, Inc., 2004.

[32] D. S. McCrickard and C. M. Chewar. Attuning notification design to user goals and attention costs. *Commun. ACM*, 46(3):67–72, Mar. 2003.

[33] K. Onarlioglu, W. Robertson, and E. Kirda. Overhaul: Input-driven access control for better privacy on traditional operating systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 443–454, June 2016.

[34] G. Petracca, L. M. Marvel, A. Swami, and T. Jaeger. Agility maneuvers to mitigate inference attacks on sensed location data. In *Military Communications Conference, MILCOM 2016-2016 IEEE*, pages 259–264. IEEE, 2016.

[35] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli. Audroid: Preventing attacks on audio nels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 181–190. ACM, 2015.

[36] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

[37] U. U. Rehman, W. A. Khan, N. A. Saqib, and M. Kaleem. On detection and prevention of clickjacking attack for osns. In *Frontiers of Information Technology (FIT), 2013 11th International Conference on*, pages 160–165. IEEE, 2013.

[38] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, Washington, D.C., Aug. 2015. USENIX Association.

[39] T. Ringer, D. Grossman, and F. Roesner. Audacious: User-driven access control with unmodified operating systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 204–216, New York, NY, USA, 2016. ACM.

[40] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[41] 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 *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[42] J. Ruderman. The same origin policy, 2001.

[43] S. Schechter. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Microsoft Technical Report, January 2013.

[44] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 51–65, May 2007.

[45] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

[46] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[47] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the eros trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 12–12. USENIX Association, 2004.

[48] M. Sheppard. Smartphone apps, permissions and privacy. *Office of the Privacy Commissioner of Canada*, 2013.

[49] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[50] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[51] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *The 20th Annual Network and Distributed System Security Symposium (NDSS), To appear*, Feb 2013.

[52] G. S. Tuncay, S. Demetriou, and C. A. Gunter. Draco: A system for uniform and fine-grained access control for web code on android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 104–115, New York, NY, USA, 2016. ACM.

[53] T. Whalen and K. M. Inkpen. Gathering evidence: Use of visual security cues in web browsers. In *Proceedings of Graphics Interface 2005*, GI '05, pages 137–144, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[54] 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 (USENIX Security 15)*, pages 499–514, 2015.

[55] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):153–186, 2005.

[56] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, pages 616–630. IEEE, 2012.

Appendices

A Compatibility Discussion

Here, we discuss how AWARE addresses special cases of applications' accesses to privacy-sensitive sensors.

Background Access: To enable background access, AWARE still uses the explicit authorization mechanism via the creation of a *binding request*. However, as soon as the application goes in the background, any on-screen security message used to notify ongoing operations over privacy-sensitive sensors is replaced with a periodic distinctive sound or a small icon on the system status bar (Section 7.1), if the platform's screen is on, or a hardware sensor-use indicator LED when the platform's screen goes off. These periodic notifications will be active until the user terminates the background activity explicitly. Our

notification mechanism leverages the concept introduced in previous work [23] and extends the mechanism used in modern operating systems for location.

Remote Access: Remote commands are instantiated by the user via an application's user interface displayed on the remote terminal, thus, the AWARE mechanisms are also applicable to the widgets displayed by such remote user interfaces. Therefore, as long as remote commands are coming from AWARE-enabled remote platforms, AWARE may pair the AWARE modules running on the two platforms by creating a Secure Socket Layer (SSL) connection to allow secure and remote control of the privacy-sensitive sensors by the user.

Programmatic Access: There are very rare cases of legitimate applications requiring programmatic access to privacy-sensitive sensors, as shown by our large-scale compatibility analysis reported in Section 8.2. Examples are anti-theft applications that capture pictures with the front camera in the attempt to identify the thief when trying to unlock the screen by guessing the passcode. Or even, an application that uses the camera to take a picture when the user smiles. However, only trusted software (as part of the operating system) should be allowed to perform such operations to be inline with our research objective of ensuring a secure use of privacy-sensitive sensors.

Hardware Peripheral Access: An application may use hardware peripherals (e.g., Bluetooth® remotes, selfie sticks, headphone jacks or built-in hardware buttons) as user interface. However, hardware peripherals are typically managed by a trusted software component, i.e., the Input Manager, and mandatory access control mechanisms (i.e., SELinux [31]) are adopted to ensure that peripheral driver files are not accessible by untrusted applications. By monitoring input events received by the Input Manger, AWARE can identify user input events coming from such hardware peripherals and bind them with the corresponding operation requests from applications.

Access through Voice Commands: AWARE enables personal assistant services that recognize voice commands, such as Apple's Siri, Google Now, and Windows' Cortana, by leveraging recent work that prevents untrusted application from exploiting voice commands by controlling access over audio channels created by applications and system services through the platform's microphone and speaker [35].

B UI Elements' Features Analysis

We performed a large-scale analysis by using the 10,000 most popular application from the Google Play store, Ubuntu Software Center and Chrome Extensions to evaluate how frequently the widgets' and activity windows' features used by AWARE change among subsequent rendering events on the platform screen. We rendered a widget and its activity window 50 times under different system settings and configurations to cause the a widget or its activity window to be rendered in different ways (i.e., screen orientation, concurrent activity windows, etc.).

ID	Width	Height	X Coord.	Y Coord.	Text Label	Text Font	Text Size
100%	99%	99%	97%	97%	100%	100%	100%
100%	*99%*	*99%*	*97%*	*97%*	*100%*	*100%*	*100%*
100%	**99%**	**99%**	**99%**	**99%**	**100%**	**100%**	**100%**

Text Alignment	Default Status	Background Color	Background Image	Border Color	Border Size	Border Padding	Transp.
100%	99%	96%	99%	99%	99%	98%	100%
100%	*99%*	*97%*	*99%*	*98%*	*99%*	*99%*	*100%*
100%	**100%**	**100%**	**99%**	**98%**	**100%**	**N/A**	**100%**

Table 4: Study of fixed features for GUI widget objects in X Window Manager, Aura (Chrome Browser) Window Manager (in *italic*), and Android Window Manager (in **bold**). The percentage values indicate how many times the widget's features did not change when the same widget was rendered by the Window Manager. We used 1,000 applications for each Window Manager system.

ID	Title Text	Title Font	Title Size	Title Color	Title Align.	Title Background
100%	99%	100%	100%	99%	100%	99%
100%	*99%*	*100%*	*100%*	*99%*	*100%*	*99%*
100%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

Width	Height	X Coord.	Y Coord.	Background Color	Background Image	Transp.
100%	100%	96%	96%	99%	99%	99%
100%	*100%*	*97%*	*97%*	*98%*	*98%*	*99%*
100%	**100%**	**99%**	**99%**	**99%**	**98%**	**100%**

Shadow	Border Size	Border Color	Border Padding	Set of UI Elements	UI Elements Position	Window Hierarch. Order
98%	100%	100%	99%	91%	99%	89%
99%	*100%*	*100%*	*98%*	*98%*	*98%*	*98%*
N/A	**100%**	**100%**	**100%**	**99%**	**99%**	**99%**

Table 5: Study of fixed features for GUI activity window objects in X Window Manager, Aura (Chrome Browser) Window Manager (in *italic*), and Android Window Manager (in **bold**). The percentage values indicate the times the features did not change when the same window was rendered by the Window Manager.

C Discussion on Activity Windows

For the ease of presentation we used the general case where a widget appears within an activity window. However, desktop and web operating system may allow more sophisticated user interfaces, or GUI scaling for different screen sizes. Thus, we recognize that an activity window could be embedded inside another activity window and such innermost activity window could be reused across several activity windows even in a hierarchy. Therefore, AWARE does not limit the use of nested activity windows or prohibit activity window reuse but rather ensures that the context is defined by the entire hierarchy of nested activity windows. As a consequence, an application may be authorized by the user to use a widget in a nested activity window X in the outer activity window Y, but this authorization does not extend for another outer activity window Z.