WindowGuard: Systematic Protection of GUI Security in Android

Chuangang Ren
The Pennsylvania State University
cyr5126@cse.psu.com

Peng Liu
The Pennsylvania State University
pliu@ist.psu.com

Sencun Zhu
The Pennsylvania State University
szhu@cse.psu.com

***Abstract*—Android graphic user interface (GUI) system plays an important role in rendering app GUIs on display and interacting with users. However, the security of this critical subsystem remains under-investigated. In fact, Android GUI has been plagued by a variety of GUI attacks in recent years. GUI attack refers to any harmful behavior that attempts to adversely affect the integrity or availability of the GUIs belonging to other apps. These attacks are real threats and can cause severe consequences, such as sensitive user information leakage, user device denial of service, etc. Given the seriousness and rapid growth of GUI attacks, we are in a pressing need for a comprehensive defense solution. Nevertheless, existing defense methods fall short in defense coverage, effectiveness and practicality.**

To overcome these challenges, we systematically scrutinize the security implications of Android GUI system design and propose a new security model, Android Window Integrity (AWI), to comprehensively protect the system against GUI attacks. The AWI model defines the user session to be protected and the legitimacy of GUI system states in the unique mobile GUI environment. By doing so, it can protect a normal user session against arbitrary manipulation by attackers, and still preserve the original user experience. Our implementation, WindowGuard, enforces the AWI model and responds to a suspicious behavior by briefing the user about a security event and asking for the final decision from the user. This design not only improves the detection accuracy, but also makes WindowGuard more usable and practical to meet diverse user needs. WindowGuard is implemented as an Xposed module, making it practical to be quickly deployed on a large number of user devices. Our evaluation shows that WindowGuard can successfully detect all known GUI attacks, while yielding small impacts on user experience and system performance.

I. Introduction

Mobile graphic user interface (GUI) system plays an important role in rendering app GUIs on display and interacting with the user, which has major impacts on the user experience of a mobile device. In particular, Android's GUI has greatly promoted user experience and gained massive popularity to the Android system. Despite the merits, Android has been plagued by a variety of *GUI attacks* in recent years.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23529

Android GUI attack refers to any harmful behavior that attempts to adversely affect the integrity and availability of GUIs belonging to other apps in order to achieve malicious purposes, such as launching a phishing or spoofing window to lure the user into taking undesirable actions, or forcefully pushing unwanted GUI content to the screen. There have been extensive recent studies on Android GUI attacks. In an attack demonstrated in [7], an attacker launches a phishing GUI immediately after an interesting event in a banking app is detected, resulting in bank account information stolen. In [2], [19], [25], the authors show that apps with certain permissions can launch different types of powerful phishing or tapjacking attacks. Notably, a more dreadful attack method, called task hijacking [28], can be done even without any permission. Specifically, by manipulating the activity browsing history saved in Android tasks, an attacker can launch a broad range of attacks, including GUI confusion, denial-of-service, and user activity monitoring attacks. Surprisingly, these attacks can affect all Android versions and all apps installed on the vulnerable devices, including the most privileged system apps and system UI.

More concerningly, the GUI attack vectors are increasingly employed by real-world malware at an alarming rate. For instance, several malware families use GUI confusion attacks to steal credit card information [35]. Ransomware, a type of malware that renders a user device useless by forcefully locking the screen until a certain amount of ransom money is paid, has migrated from PC to the Android world [4], [26], infecting more than 900 thousand Android devices within two years [9]. In addition, adware, which repeatedly presents unwanted (and sometimes "unclosable") advertisement windows to the user, is not only irritating, but also makes the user prone to further malware infection [27], [33]. Given the severity and rapid growth of GUI attacks, we have a pressing need for a comprehensive defense solution to effectively mitigate the emerging threat.

Challenges: A recent defense solution has been proposed in [2], which involves a two-layer defense: an app vetting process based on static analysis, and an on-device defense mechanism. The static analysis scans for the suspicious use of GUI-related APIs/permissions and flags the apps who use them as malicious. A fundamental challenge (also mentioned by the authors) is that it is difficult for code analysis to determine the real purpose of using these APIs/permissions. For instance, although a legitimate screen locker app may use the same set of APIs and exhibits similar behavior (launching a lock screen) as a malicious ransomware, the purposes of the two apps are totally different. In this case, it is up to the user to decide if a screen lock is desirable or not based on the runtime context. User involvement is adopted by the second on-device defense in [2], which inherits the idea of Extended Validation Certificate (EV) green address bar in browsers. By putting a reliable app origin indicator in the navigation bar, an app can constantly inform the user of its identity as long as the origin indicator is visible. By this means, it prevents user from providing sensitive information to the wrong entity in a GUI confusion attack. Despite the novelty of this solution, the passive defense solely relies on the correct judgment of savvy users and thus requires users' continuous attention to the indicator on the navigation bar. This not only largely affects user experience, but also undermines its effectiveness, e.g., reportedly only 76% detection rate at best in a user study. Moreover, this defense strategy is only helpful in GUI confusion attacks, but cannot defeat other types of GUI attacks, e.g., denial-of-service attack. In addition, the requirements of modifying both existing apps (implementing HTTPS EV certification) and the system make it impractical to be adopted by any significant portion of apps or systems in the Android ecosystem.

Our approach: It is exactly these challenges that this paper seeks to address by proposing a viable new solution. We take the first steps to systematically scrutinize the security implications of Android GUI system, one of the most sophisticated subsystem in Android. At the heart of the problem, a GUI attack occurs when an attacker interferes with the normal user session such that the attacker's GUI finally takes over part or all of the device's display (regardless of if the user realizes it or not). Although the Android security model renders different apps sandboxed and isolated from one another, the user session - a series of GUIs that a user has visited when doing a particular job - is typically a joint "effort" from different apps, and is beyond the protection scope of the existing security mechanisms. The problem is further exaggerated by the uniqueness of mobile GUI environment. That is, given the lack of app identifier and user control on the screen, plus a plethora of APIs that can be abused to affect the GUI system states, the normal user session is extremely vulnerable to arbitrary interruption or manipulation by the attacker, e.g., popping up a phishing window on top of the current app, modifying an app's window history, etc.

To fill this important security gap, we propose a new security model - Android window integrity (AWI) - to comprehensively protect the system against GUI attacks. AWI is a generic security model that clearly designates a user session, specifies the capabilities of various other principals in the system, and defines the legitimacy that the GUI system should keep from one state to another. AWI is carefully designed to conform to the Android app model and the norm of app navigation. By doing so, it protects the normal user session against abrupt interference by other apps while still preserving the original user experience. Our implementation of the model, namely WindowGuard, can systematically protect the GUI system and aims to defeat all GUI attacks, a much broader range of attacks than previous work. When WindowGuard is deployed on user devices, the user is not bothered at all until a suspicious behavior is detected, caused by the violation of a set of integrity criteria defined in the AWI model. WindowGuard then briefs the user about the security event and asks for the final decision from user, who is inherently more capable of making the best decision for him/herself based on the context. More importantly, this design makes WindowGuard more usable and practical to meet diverse needs from users and app developers in the current Android ecosystem. Our evaluation shows that WindowGuard can immediately detect all known GUI attacks with minimal performance overhead. We also evaluate the usability of WindowGuard over 12,060 most popular Google Play apps. We find that the WindowGuard has no usability impact on most apps. Among the 1.03% of apps that trigger the security alert, most of them are only involved in one type of security enforcement, which can be promptly turned off for that app based on user decision and will not distract the user any longer after that. In summary, we make the following contributions:

- *New understanding of the Android GUI security.* To the best of our knowledge, we are the first to systematically overhaul the security implications of the Android GUI system design, a complex subsystem that is composed of a variety of system services and components. This new understanding can further inspire follow-up research on mobile GUI system security.
- *Novel GUI security model in mobile environment.* We propose a novel security model - Android window integrity - for the GUI subsystem in a unique mobile environment like Android. By clearly specifying the capabilities of various principals in a user session and defining the legitimacy of GUI system states, AWI is able to comprehensively and automatically protect a normal user session against a wide spectrum of GUI attacks. More importantly, the new security model also considerably raises the bar for future attacks. New attacks can now be put under the test of our defense before they cause real threats.
- *Implementation and evaluation.* WindowGuard is developed as an Xposed module that can be quickly deployed and protect a large number of user devices. WindowGuard implements the AWI model and engages the user when suspicious behaviors are detected. This design involves user input in context-aware "block it or not" decision-making and also preserves desired user experience based on user's choices. Our evaluation shows that WindowGuard can detect *all* known GUI attacks while yielding small impact on device usability and system performance.

II. ANDROID GUI SYSTEM

Android GUI subsystem is composed of various system services and components and requires a close collaboration of them. *Activity Manager Service* (AMS) and *Window Manager Service* (WMS) are among the most important ones. In this section, we introduce how Android GUI system works and identify the security risks of this complex system.

A. Activity and Window

Activity: Activity is a type of app component that provides one or more windows to the user. Activity and window are

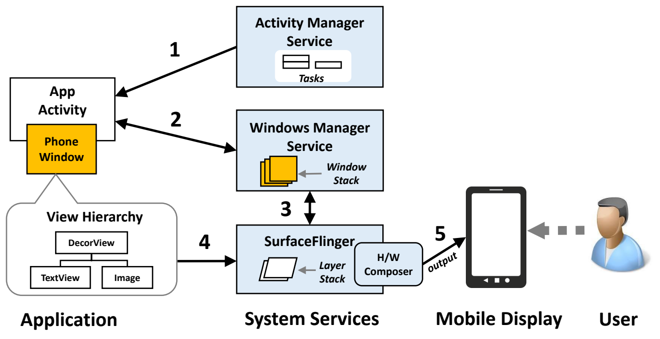


Fig. 1. Overview of Android GUI System Architecture

closely related: each activity must have a window instance, which contains the GUI contents to be displayed. The GUI contents are specified by the app developer. If not explicitly specified, a default GUI will be populated to the activity's window. In Android, activity is designed to be the building block of app navigation. For example, when an user opens an email app, the first activity may show a login screen; after that the second activity displays the inbox; and then the user opens the third one to compose a new message.

Window: Conceptually, window is a visual area on screen that shows the GUI of the program it belongs to. In an app activity, a window instance is a container that holds hierarchical GUI elements (view hierarchy) to be displayed on screen. In Window Manager Service, each window is represented by a *WindowState* instance, which contains all the parameters about the window such as size, location, transparency, and z-order etc. There are usually multiple windows at the same time and the system composes them into one frame to be displayed on screen. Android system defines three classes of windows: (1) App window: the top-level window that is associated with an activity; (2) System window: a set of windows that are used for special purposes, e.g., status bar, navigation bar, and key guard etc. (3) Sub-window: the window that is attached to the other two classes of windows. It is noteworthy that although system windows are mainly used by the system, third-party app can also launch system windows as long as it is granted certain permission (as we will see in Section II-B). For example, method input or wall paper windows could come from third-party packages.

B. GUI Architecture Overview

Figure 1 shows the architecture of Android GUI system and illustrates how an app's activity manages to display its window on screen. The process involves the following steps from a high level: (1) AMS launches a new activity of an app. (2) The new activity then creates a window for itself during initialization and registers the window to the WMS. WMS creates a new WindowState instance that represents the new window and add it to a window stack, which contains all available WindowState instances in the system. (3) On behalf of the app, WMS then asks the *SurfaceFlinger* to create a drawing surface for the window, i.e. a buffer containing graphical data. The drawing surface, also known as *layer*, is in turn shared with the app by passing a handler back to the app. Meanwhile, WMS also provides the window parameters to SurfaceFlinger such that the latter can later use this information to compose the final frame. (4) Once the app receives the shared surface, it can start drawing the window's entire view hierarchy on the surface and signals SurfaceFlinger when the drawing is completed. This drawing process is synchronized with SurfaceFlinger and can happen as fast as 60 frames/second. (5) SurfaceFlinger keeps multiple layers of different windows in a *layer stack*. When all the visible layers are ready, SurfaceFlinger composes them and displays the final frame on the screen, with the help from *Hardware Composer* (HWC), a device-specific Hardware Abstract Layer (HAL) library.

In this architecture, although SurfaceFlinger has direct control to the display hardware, it strictly carries out the commands from its "supervisors". The "supervisors", i.e. AMS and WMS, not only control how the windows should be displayed, e.g. window size, location, transparency, z-order, but also determine what windows should be made visible. In other words, these two system services are the heart of Android GUI system. We now introduce them by looking at an example shown in Figure 2.

1) Activity Management: Activity management is performed by AMS. Activities are started by AMS upon client requests through *intents*, an abstract description of the activity to be started. As shown in the left part of Figure 2, every app activity has one-to-one mapping to its corresponding *ActivityRecord* instance in AMS. For convenience, we refer to an ActivityRecord instance in AMS simply as activity for the rest of this paper.

AMS organizes all activities in *tasks* [1]. Each task includes a stack of activities, namely *back stack*. Activities in a back stack are ordered by the time that they are visited, such that the user can go back to the most recent activity. There is only one activity running in the system at a time, called *focused activity*. The task that contains the focused activity is *focused task*; all other tasks are in the background. Figure 2 (in AMS) shows multiple tasks: two for app A and B, respectively, and another task for the launcher. When the user clicks an app icon in the launcher, a new task is typically created and the main activity of the app becomes the root activity in the task. When another activity is later started, it is by default pushed on top of the task that launched it. For instance, activity A2 is started by A1 and is thus put on top of task A. Activity A1 is stopped but remains in the task, whereas A2 gets the focus and is shown on display. When the user later presses the back button, the top activity is popped from the task and destroyed, and the next activity A1 on stack is then resumed and become the focused activity again. Since activity is the building block of the application model in Android, AMS essentially controls the app navigation, and the overall user experience in Android.

2) Window Management: WMS is responsible for a variety of jobs. One of the most important one is to manage all windows in the system, update their parameters and pass them to SurfaceFlinger when needed. Specifically, each window is represented as a *WindowState* instance in WMS. For convenience, we refer to a WindowState instance simply as a window for the rest of this paper.

All windows in a display (usually only one for mobile device) are kept in one stack, namely *window stack*. Once changes happen to any window or the window stack, e.g., window re-sizing, activity launch, etc., WMS would walk

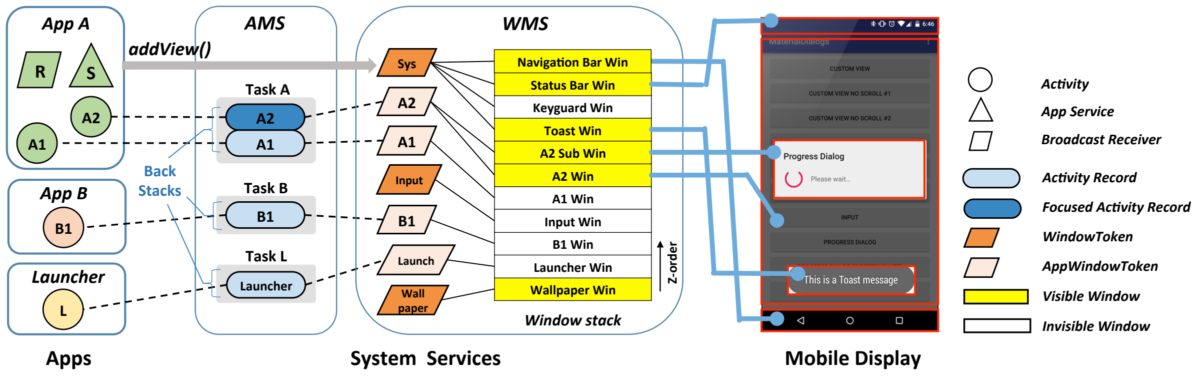


Fig. 2. A snapshot of system states in AMS and WMS

through the window stack, calculate the new parameters for all windows, and pass the new information to SurfaceFlinger, which would compose all updated layers and finally reflect the changes on screen.

As shown in Figure 2, the window stack ranks windows by a numeric value called *z-order*. A window with higher z-order resides higher in the stack, which means that the window will appear on top of others if window overlap occurs. A preliminary z-order is first assigned to a window based on its type. For example, app windows have lower z-order values while system windows (such as status and navigation bar) have higher values and hence are shown on top of screen. The final z-order is further determined based on a variety of factors, such as window function, order of creation, etc. For instance, the z-order of an input method window is always set to be a bit higher than the app window (B1 window) that requests the input keyboard, but not higher than the window that is originally above the app window.

A window can either be visible or hidden, and there are usually multiple visible windows simultaneously. Figure 2 exhibits the visible windows in WMS and their window areas on screen. Here, visibility does not refer to whether being visible by naked eye. Instead, visibility is a window state. if "visible", it indicates that the window is ready for display. Nevertheless, whether the window will finally be shown on screen depends on the location, transparency and z-order of other "visible" windows. That is, if the window area is completely overlaid by other windows higher in the window stack, the window is in fact hidden "behind" the foreground windows even if the state is "visible". For example, although the visibility of the launcher's wallpaper window in Figure 2 is always set to be "visible", it is completely overlayed by other visible windows (because it resides at the bottom of the window stack) and is in fact invisible on display. On the other hand, if visibility is set to "hidden", a window will not be displayed at all.

Given the windows in window stack, WMS is able to quickly and reliably identify the owner of each window by using *WindowToken* and *AppWindowToken*. As its name implies, a WindowToken is a type of binder token (will be discussed in Section II-C) that is used to uniquely identify a group of related windows in the system. AppWindowToken is similar to WindowToken, except that all windows in the group are associated with the same activity. In other words, an AppWindowToken is a representation of its corresponding activity instance in AMS. Given these two types of tokens, we classify all windows in the system into two categories. As we will see, it is important to make such distinction for the design of GUI security scheme.

1) Activity window: the window that is associated with an activity. It may be a top level app window or a sub window attached to the app window. An activity may have multiple windows. These windows are grouped into a list under the corresponding AppWindowToken, which uniquely identify the activity in AMS.
2) Free window: a window that is not associated with any activity. A free window is either a system window or a sub-window attached to the system window.

Figure 2 shows all available WindowTokens, and AppWindowTokens including their one-to-one mappings to ActivityRecord instances in AMS. Notably, two activity windows are visible and both belong to token A2: one sub window in the foreground (a progress dialog "A2 Sub"), and another app window ("A2 Win") beneath it. They are emplaced above all other activity windows by WMS when activity A2 becomes the focused activity. In addition, an activity window is always started by an activity. On the other hand, both the system and third-party apps can directly launch free windows by making an `addView()` API call to WMS, e.g., app A is free to start a toast window (one type of system window) on top of other windows in Figure 2.

C. Security Mechanisms

There are three security mechanisms that play critical roles for GUI security in Android: app sandboxing, binder token, and permission scheme.

In Android, every app is given a unique Linux UID and runs in a separate process by default, which effectively isolates one app from the others. From GUI's perspective, sandboxing guarantees the isolation of the graphic information in each app window, e.g., preventing an app from modifying the drawing surface of another app, given that the system itself is not compromised.

In reality, an app crosses process boundaries and communicate with system services to enable its proper functionality by using an important IPC mechanism called *binder*. In this client-service communication, it is crucial for the system services to securely identify who the client is. It turns out that a binder object has a unique property that is supported by the *binder driver*: each binder object maintains a globally unique identifier across all processes. This property makes a binder object ideal for app authentication purpose, i.e., a binder object is used as a security token shared between client and system service. Specifically, AMS and WMS create app/window tokens (e.g. AppWindowToken and WindowToken) and share them with a client app. When a client app (or its enclosed activity) asks for service, the system services require the client to identify itself by including the given token in its request. This authentication scheme is a cornerstone of Android security architecture. It prevents system services from being spoofed by the attacker, e.g., mistakenly modifying the window states of another app, or delivering user inputs given on a window to the wrong app.

Moreover, Android provides finer-grained security enforcement based on its permission mechanism. Apps must explicitly declare the permission they need for additional capabilities. For example, an app can launch certain system windows only if it has obtained the `SYSTEM_ALERT_WINDOW` permission. Permission can also be declared to protect activity. During activity launch, AMS enforces access control by consulting *Package Manager Service*, which checks the permission requirement of the callee activity (if there is any) and returns the check result to AMS.

D. Security Risks

Although the current security model works especially well in many aspects, it performs poorly in protecting the GUI system against existing attacks. The fundamental problem is, in the GUI context, it is the user session - a list of GUIs that an user has visited in a time series when doing a particular job - that requires security guarantees and protection. The security of app sandboxing is only enforced on process/app boundaries, leaving the user sessions vulnerable to being arbitrarily interrupted or manipulated by attacker, e.g. popping up a phishing window, or redirecting the user to a different task during app navigation.

Unfortunately, this problem is further exaggerated by the uniqueness of GUI in mobile environment. First, unlike the desktop machines, the screen of a mobile device is relatively small and usually only shows one app at a time. To save space, there is generally no app identifier on screen, e.g. a task bar or a window title bar like in desktop OSes. Because of this limitation, it is cumbersome for a user to constantly verify the real identity of the current GUI. Although a user can resort to the recent button for the name of the current task, the displayed information is untrustworthy as the recent task list is subject to manipulation by attackers [28]. Therefore, for a normal user reliably identifying the current GUI becomes infeasible. Secondly, the GUI design and the norm of app navigation makes GUI attacks more likely to succeed. For example, because the user has been accustomed to the default app navigation behaviors, one might be easily spoofed if these default behaviors are maliciously tampered. Due to the lack of user control to the screen, it is troublesome for the user to escape a lock screen (e.g., a non-escapable system window) in a denial-of-service attack. Thirdly, Android provides app developers with great flexibility to control the window states in the system without being strictly disciplined. For example, a normal app can freely launch new activities, add high z-order system windows on screen, or modify other apps' back stacks. Originally intending to promote the platform features for app developers, these features inadvertently enrich the GUI attack vectors and facilitate the mal-behaviors. In fact, Google has long realized the security issues of the over-flexible GUI features, and has taken steps to remedy the problems in newer Android releases, e.g. adding security attributes to GUI components, requiring explicit user consent to certain permissions, enforcing runtime permission, etc. However, many security features are barely used by unwitting developers (even Google apps themselves). Even if they are fully employed, the ad-hoc protection cannot systematically mitigate all attacks. On the other hand, removing or modifying the longstanding GUI features will break a bulk of existing apps.

Category	Attack Vector	Consequences
Window Overlay	UI interception attack [13]	*A*
	Tapjacking attack [19], [25], [29]	*B*
	Toast message [19], [25]	*A, B*
	Phishing attack [2], [7], [13], [29]	*A, C*
	Immersive full screen attack [2]	*A, C*
	Denial of Service	*D*
	Adware	*D, E*
Task Hijacking	Back button hijacking [2], [28]	*A, C*
	App launch spoofing [28]	*A, C*
	Denial of Service [28]	*D*
	User monitoring attack [28]	*F*
	`moveTaskTo` APIs [2]	*A, C*

TABLE I. EXISTING KNOWN GUI ATTACK VECTORS IN PRIOR WORK. THE CONSEQUENCES ARE: *A* - SENSITIVE DATA STOLEN; *B* - USER INPUT EAVESDROPPING; *C* - USER SPOOFING; *D* - LOSS OF AVAILABILITY; *E* - MALWARE INFECTION; *F* - USER PRIVACY INFRINGEMENT

As a result, the Android GUI system becomes particularly vulnerable to a variety of GUI attacks that can be easily launched without confinement. This is proven by both the prior research findings and the rapidly growing real-world threats. To the best of our knowledge, Table I shows a full list of known GUI attacks. Depending on attack vectors, all attacks are classified into two categories:

- *Window overlay attack*: attacks that render a window on top of screen, partially or completely overlaying other windows.
- *Task hijacking attack*: a class of malicious behaviors that trick the system to modify the app navigation behavior or the tasks (back stacks) in the system.

Both categories of attacks can cause serious consequences as shown in the last column of Table I. In summary, we consider a threat model as follows:

Threat Model: We consider a harmful app has been installed on the user's Android device. Like most real-world malware, the harmful app does not have system privilege, e.g., running with a system UID. Instead, it may seem harmless, requiring a minimal set of permissions needed for the malicious purpose. We assume that the system itself is un-compromised and trustworthy. We also assume that one window involves one principal. In the cases when a window is composed of elements from different principals, e.g., the app itself and an embedded third-party ad library, we consider the principal to be the owner of the window. In order to achieve its malicious purpose,

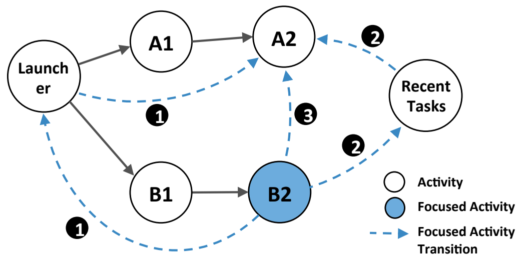


Fig. 3. Multiple activity sessions in the system.

the attacker's goal is clear: affect a normal user session to the attacker's advantage using windows or activities under attacker's control.

III. Android Window Integrity

As discussed, the existing security model is not designed to cope with GUI attacks. The fundamental limitation is, GUI attack targets a normal user session, a series of windows that user has visited, which is beyond the scope of app sandboxing protection. Motivated by the challenges and serious threats, we propose a new security model - Android Window Integrity (AWI) - to fill in this important security gap. AWI is a generic model for Android GUI security. It redefines a user session as a chain of activities starting from the launcher, specifies the capabilities of various principals in the system, and describes the criteria of how the GUI system should be kept valid from one state to the next.

A. Display Owner and Activity Session

The key principle of AWI is that, no application, by default, has permission to perform any operations that would adversely affect the user session of other apps or the system UI. At the center of the model are *display owner* and *activity session*, which are the basic entities to be protected in AWI, just like an app process in the app sandboxing.

As discussed in Section II-D, mobile display is an unique time sharing resource that is shared by different apps at different time. We introduce display owner, the one and only one principal that is more privileged than other apps and "owns" the mobile display at a time. In AWI, we specify the *display owner* to be the app of the currently focused activity. It means that the app of the focused activity is more privileged than others in terms of GUI-related operations, and its windows and user session is protected under the AWI model (although the display owner is still disciplined by the existing security mechanism).

In Android, app navigation always starts from the launcher activity, the first focused activity after system boot-up and the primary app navigation "hub". A user starts a job by opening an app activity from the launcher and later proceeds to other activities as the job goes on. The states of previous activities in a job are saved and can later be resumed by tapping the back button. Once the current job is finished, one can go back to the launcher via the home button, or switch to another job by going to the recent task list (pressing the recent button), another navigation "hub".

AWI complies with this norm of app navigation by introducing activity session. An *activity session* is a sequence

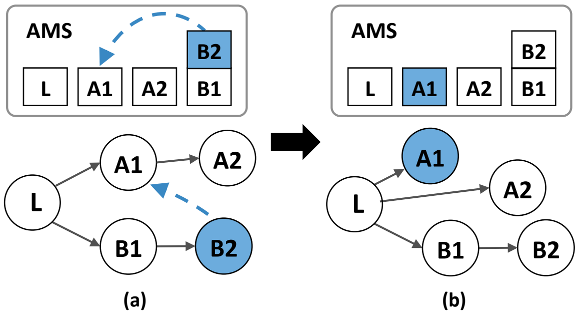


Fig. 4. An activity session transition example.

of activities that guide the user from the launcher activity to the currently focused activity. It intends to capture the user's visual experience of previous activities, the basic building blocks of an app, when doing a particular job. Specifically, as shown in Figure 3, each node indicates an activity. An activity session always starts from the launcher activity. When a new activity is displayed on screen, it is chained to the end of the current activity session. There can be multiple activity sessions at the same time. All the activity sessions form a tree: the launcher activity is the root and the only joining point of all activity session branches. The focused activity (belonging to the display owner) is always at the tail of an activity session, which is called *focused activity session* (e.g. $launcher \rightarrow B1 \rightarrow B2$ in Figure 3).

The user can switch to another activity session anytime, called activity session transition. For instance, as illustrated in the diagram of Figure 3, focused activity B2 may transit to A2 following three routes (the dashed line). In 1 and 2, the user can either go to the launcher or the recent task activity, and then resumes A2. In particular, the recent task activity belongs to system UI and is only started upon user pressing the recent button. Since its function is to be a "hub" that facilities task switch in the system, we regard the recent task activity itself as a separate activity session that does not overlap with any other activity sessions. In this sense, the transition from B2 to the recent task activity is an activity session transition by itself. In route 3, a focused activity transition can also occur directly from B2 to A2. This can happen in many circumstances, e.g., launching an existing activity with special intent flag, etc.

Although an activity session looks similar to a task in AMS, the two are independent and have major differences as we will see in Section III-C. Note that the sequence of activities retained in an activity session may be saved in multiple tasks/back stacks by AMS. Figure 4(a) shows such an example, in which both the activity session and corresponding system state are depicted. Activity A2 was launched by A1 but placed into a new task in AMS (e.g. by specifying *singleTask* activity attribute). Since A1 and A2 are displayed in a sequence, they are regarded to be in the same activity session. If the focused activity then transits from activity B2 to A1 as a result of any of the above reasons, the original activity session will be divided into two sessions (still rooted at the launcher activity), as shown in Figure 4(b).

Given these two important notions, we next introduce the three aspects of legitimacy that AWI attempts enforce on the system states. But before we proceed, to better understand these principles, we first simplify the complex GUI system and formalize it in a state transition model.

B. System State Transition

The state transition of the GUI system is described by $(S, \Lambda, \rightarrow)$, where $S = \{S_{AMS}, WS\}$. S denotes the set of system states; S_{AMS} indicates the set of system states in AMS; WS stands for the set of window stack states in WMS. Λ indicates a set of events and conditions that invoke a set of viable transition $\rightarrow$. Specifically, given a system state $s \in S_{AMS}$ and a window stack state $ws \in WS$, they have the following states:

- $s = \{a_{focus}, \beta\}$, where $\beta = \{bs_1, bs_2, ..., bs_n\}$, and $bs = (a_1, a_2, ..., a_m)$. a_{focus} denotes the current focused activity. β is a set of all n back stacks in the system. bs denotes a back stack, each in turn includes an ordered list of activities, indicated by $(a_1, a_2, ..., a_m)$.
- $ws = (w_1, w_2, ..., w_n)$. ws represents the window stack containing a total of n windows. w_1 and w_n represent the bottom and the top windows in the stack respectively. Each window w_i $(1 \leq i \leq n)$, includes a few parameters such as visibility, size, transparency, etc.

C. System State Legitimacy

Android window integrity is composed of three types of legitimacy: the legitimacy of the past activity session, the legitimacy of the current visible windows, and the legitimacy of the future windows to be displayed. Instead of being a rigid security model, AWI adapts to the diversity of user needs by incorporating user's choice in the model. Once an integrity violation is detected, it is up to the user to make the final decision. This makes AWI both usable and practical to be employed in reality.

Legitimacy of activity session. An activity session looks similarly to a task's back stack at the first glance, e.g. both keep a record of previous activities. However, activity session is not a simple duplication of task. A task is a container that keeps existing activity instances. Although the order of activities in a back stack is typically reserved and follows the order of activity launch most of the time, it is not always true. In fact, Ren et. al. [28] demonstrated that the back stacks could be manipulated outrageously by abusing the task features, e.g., an activity could be relocated to other tasks; app navigation behavior could be changed; a full back stack of activities could be created without user's awareness. This flexibility however contradicts with user's common sense, e.g., it is commonly believed that back button pressing should resume activities that the user has previously seen; clicking an app icon from the launcher ought to start the app window (which may not be the case), and so on. By taking advantage of these pitfalls, the most dreadful task hijacking attacks could be launched, as listed in Table I.

On the other hand, activity session is designed to simulate and preserve user's visual experience by saving the sequence of visible activities when the user is doing a job. Every time the focused activity changes, the foreground activity session is used to check the integrity of the back stacks by comparing the difference between the two. Any disparity between the two indicates a mismatch of user's visual experience and system state, and is considered suspicious. More specifically, the model considers the following statement as a proper system state:

$$\exists \{bs_1^*, bs_2^*, ..., bs_n^*\} \subseteq \beta : s_{fg} = (bs_1^* \parallel bs_2^* \parallel, ..., \parallel bs_n^*)$$

where β indicates the set of all back stacks, in which bs_i^* is one of them. s_{fg} denotes the foreground activity session, composed of an ordered list of activities. In other words, it checks if there exists a subset of back stacks, such that the concatenation of their ordered activity lists is the same as the ordered activities in the foreground activity session.

The failure of activity session integrity check may indicate possible attack, but it may also come from the use of task features for legitimate purposes. The distinction lies in an important premise of task hijacking: *a task hijacking attack happens only if the malicious activities manage to reside in the same task together with the legitimate activities.* Given this premise, AWI iterates the back stacks that are part of the focused activity session. If the activities in the back stacks are all from the same app, AWI regards it to be valid. Otherwise, a notification is created to alert the user about the event and possible security hazard.

Legitimacy of current visible windows After an activity gets focus and becomes the display owner, other than its own activity window(s), there are usually other visible windows in the window stack. To prevent the display owner's activity window from being disturbed by unauthorized windows from other apps, an overhaul of the window stack is necessary. Specifically, the model specifies that no other visible windows, except the windows belonging to the display owner app and a set of white-listed windows, should overlay on top of the focused activity window in the window stack. To put it formally, the model have the following guarantees:

$$\neg \exists w_i \in ws : w_i.visible = true, i > k,$$
$$w_i \notin FocusedApp, w_i \notin L$$

where w_i is the i^{th} window from the bottom of window stack; k represents the index of the top focused activity window in the window stack; and L denotes a white list of windows. The white list typically includes system windows and other windows explicitly specified by the user, as we will discuss in the implementation in Section IV.

Legitimacy of future windows. There is a plethora of approaches to launch a window in Android, but all windows are in two categories: free window and activity window. The legitimacy of future window is defined as: given the current display owner, the principal (identified by UID) that initiates the launch of a new window must be either the display owner app itself or from a white list specified by the user. This criteria holds for both free window and activity window. We now explain the two cases respectively.

Although third-party app or package can launch free windows, such as toast window (without any permission), or other system windows (requires `SYSTEM_ALERT_WINDOW` permission), many free windows are typically launched by the system or system apps. The model considers a white list of UIDs of the system processes (e.g. system server) and system packages (e.g. system UI) trustworthy, and allow their windows being displayed freely. Moreover, there are two types of windows that the system treats differently: input window (including input method or dialog windows), and wallpaper window. An input window is registered in the Input Manager Service. When a window requests input method, the Input Manager Service launches the input window on top of the client window in the window stack. Wallpaper window is similar, except that it is started by a wallpaper service and is placed under the client window. Since the type and the client window of an input or wallpaper window are securely specified by the system, they are considered trustworthy (as long as the input method or wallpaper packages themselves are legitimate). The model allows input or wallpaper windows being started as long as their client window is one of the currently visible windows.

When an activity window is started, either (1) the focused activity remains the same (e.g. launching another top-level app window or a sub-window) or (2) the focused activity changes, and the activity window belongs to the newly focused activity. The first case is always valid because the display owner is unchanged and the activity window must be started by the display owner app (assured by the security guarantee of app window tokens). In the second case, the change of focused activity implies a possible change of display owner. Given the flexibility of Android APIs and task features, there are numerous possibilities that would result in the change of display owner, which we will characterize in Section IV-A. Despite the complexity, the same principle still holds, i.e., the change of focused app must be initiated by the display owner app or white-listed principals.

Among the many possibilities of focused activity change, back button pressing is special. Back button is one of the most popular user navigation control always available on screen. By default, one can navigate backward to the previous screens by pressing the back button. In this case, the system destroys the current focused activity and resumes the next activity on back stack. However, the back button behavior can be changed by overriding the *onBackPressed* callback function of the focused activity. Although this flexibility is useful in many cases, e.g., fragment or webview navigation within the same activity, it is sometimes confusing to the user if being mis-used, e.g., instead of "going back", it re-directs the user to some other activity. The model does not regard the customized back behavior malicious as it is defined and initiated by the focused activity. That being said, the model keeps an eye on the program behaviors after a back button pressing, and raises a toast message when user confusion is possible.

IV. WindowGuard

We implement the AWI as a module for the Xposed framework, a popular code-injection framework for rooted Android devices. The implementation, namely *WindowGuard*, is tested on Google Nexus 5 phone and can be used in Android 4.4, Android 5.x and Android 6.0 with minor changes. An Xposed module can hook arbitrary functions of the system or apps at runtime and change their behaviors without modifying the system or apps themselves. WindowGuard can be used on all Android device brands that the Xposed framework supports. These features makes WindowGuard practical to be distributed to a large number of Android devices and provide immediate protection.

A. AWI Model Implementation

WindowGuard implements AWI by hooking 26 functions of AMS, WMS, Package Manger Service (WMS), and system UI in a total of 2300 lines of code.

Activity session integrity. As previously depicted in Figure 3, activity sessions are implemented as a tree data structure maintained in the *system server* process (a privileged process hosting all system services implemented in Java). Each node represents an activity, which links to its predecessor and successor activities. Activity sessions share the same root, the launcher activity. A new activity node will be added to the current foreground activity session only if a new activity obtains the focus (its windows become visible), and is destroyed together with its corresponding activity. WindowGuard hooks functions in AMS to perform back stack integrity check upon the change of focused activity, as discussed in Section III-C.

Access control of free windows Given a display owner, access control is enforced on the free windows that are about to be displayed or resumed based on the discussion in the legitimacy of future windows. For those existing visible windows that violate the window legitimacy criteria, free windows are made invisible, and activity windows (if there is any) are reordered under the focused activity windows in the window stack. To achieve this, WindowGuard hooks a handful of WMS functions performing functions such as adding windows to the window stack, window stack reordering, and window visibility control. Access control of free window helps prevent window overlay attacks such as user spoofing or a ransomware denial-of-service attack. In a denial-of-service attack, even if the ransomware manages to "lock" the screen using a free window, the use can always click home or recent activity button to escape the lockscreen.

Safeguarding focused activity transition Focused activity transition happens frequently during the use of device in practice. It happens either during activity launch or resumption and may result in a new display owner. In principle, the focused activity transition must be initiated by the display owner app in either cases, as previously discussed. Due to the complex app navigation behaviors, the focused activity transition requires close monitoring of a variety of system functions or APIs.

A new activity gets focus when it is launched by one of the *startActivity* function calls from an app. The origin of the caller app is examined. If the caller app UID is not the current display owner or one of the white-listed principals, an alert notification with detailed information is prompted to the user for confirmation to proceed. This effectively prevents attacker from overlaying an malicious activity window on top of a victim app. Similarly, resumption of existing activity can also trigger focused activity transition. Activity resumption could occur either passively or actively. For example, the next activity on the back stack is passively resumed when the current focused activity quits and is destroyed by the system. It is perfectly valid in this case for the display owner to yield its own privilege. Activity resumption can also be actively initiated by invoking a set of APIs such as *startActivity*, *moveTaskToFront/Back*, or *moveTaskForward/Backward*. Specifically, *startActivity* can resume an existing activity under certain conditions, e.g., the activity's launch mode is *singleInstance*. The latter two APIs can move a task to the foreground or background, which essentially changes the display owner at will by any app as long as *REORDER_TASKS* permission is granted. WindowGuard hooks the internal functions in AMS for each of these app APIs to check the caller origin. A user confirmation is further requested if the caller origin is not from the display owner.

B. Security of App Navigation "Hubs"

The launcher and recent task list play crucial roles and act as app navigation "hubs". Due to the vendor customization or availability of third-party alternatives, the security implication of these components is unknown. In addition to the security provided by the generic AWI model, WindowGuard provides extra protection on these critical components.

Launcher: Launcher is the first app activity to be started. Other than the default launcher that comes with the stock system, third-party launchers are also available. Our first goal is to securely start the launcher activity of user's choice. After system boot-up, AMS queries PMS about the information of packages that can serve as the launcher. If multiple launcher activities are returned, a dialog is prompted for user to make a choice. At this stage, WindowGuard is able to prevent an attacker from affecting user's choice by manipulating windows on screen, as the display owner is system UI (who owns the dialog activity). WindowGuard trusts the user's choice and regards the chosen launcher activity as the only root of all activity sessions in this system launch.

The second goal is to assure that an app is reliably started when the user clicks its icon in the launcher. Here we discover a security issue in the app launch process, which affects all Android and launcher versions. When an app icon is clicked in the launcher, an intent with *NEW_TASK* and *ACTIVITY_BROUGHT_TO_FRONT* is sent to start the corresponding app. The combined use of these intent flags creates a new task to host the app's new activity. If the app's task already exists, the task will simply be brought to the foreground. However, in this operation, AMS considers the task owner to be the package name described in the *taskAffinity* attribute of the root activity (the bottom activity in the back stack), instead of the app of the root activity itself. Although the two are by default the same, the *taskAffinity* attribute can be configured arbitrarily to some other app's package names without restriction. Therefore, a malicious activity can spoof the system by specifying a victim app's package name as its *taskAffinity*, and start the activity in a new task. The task is then believed by the system to be the victim app, but in fact, is controlled by the attacker. The problem occurs when the user clicks on the victim app, yet the malware task is started instead of the victim app itself. To remedy this problem, WindowGuard monitors the requests of activity launch in AMS. If it comes from the launcher, WindowGuard saves the app to be started, and later verifies that if the focused task indeed belongs to the app by checking the origin of the task's root activity.

Recent Task List: The recent tasks screen contains a list of all recently accessed tasks, and for each task it shows the task owner's name/icon and the task's last screenshot. The user can browse through the list and choose a task to resume. However, the recent task list suffers from the similar problem of task ownership confusion, because it regards the owner of a task to be the app described in the *taskAffinity* attribute of the task's root activity. As a result, the user could be easily spoofed by a malicious task which camouflages as the victim app in the recent task list. To impede such an attack, the system UI is hooked such that it shows the name/icon of the root activity app of a task, instead of what is described in the *taskAffinity* attribute. By this means, it faithfully reflects the real identities of tasks in the system.

C. Preserving User Experience

WindowGuard implements the AWI model, which is designed to adapt to the Android use and navigation pattern, such that the user experience is not affected at all in normal use until a security violation is detected. The security violation may indicate a potential GUI attack or a legitimate use of GUI features that do not strictly follow the norm of Android app model. WindowGuard takes a light-weight response by briefing the user and asking for the user's final decision upon a security event, such as block, allow for once, or add to white list, etc. The alert messages, depending on the emergency and severity of attacks, are delivered via a confirmation dialog, a system notification or a toast message right after the violation occurs. WindowGuard maintains a handful of white lists; one for each security feature. Those on a particular white list are not confined by the corresponding security feature. WindowGuard always respects the user's decision and the diversity of user needs; the white lists are promptly updated based on user input. Moreover, all GUI security protection features can be lifted and re-enforced in a centralized control panel, making it convenient for the user to tune the security features based on preference.

V. EVALUATION

We now proceed to the empirical evaluation of the efficacy of WindowGuard in the following facets: effectiveness, usability and performance impact.

A. Effectiveness

To evaluate the effectiveness of our solution, we install the WindowGuard prototype on a Google Nexus 5 phone and experiment with 15 attack samples from all 12 attack vectors listed in Table I. The attack samples are either real-world malware/adware, or are proof-of-concept apps we developed based on previous research [2], [7], [13], [19], [25], [28]. The evaluation shows that WindowGuard is able to effectively detect and defeat all attacks. We now show a few case studies to demonstrate how the attack behaviors violate AWI and how WindowGuard delivers the potential attack alert to the user.

Back button hijacking: Back button hijacking [28] is one type of task hijacking attacks. The attack misleads the user to a phishing activity after the user clicks the back button, instead of the original activity the user just visited. Figure 5 shows the task states in AMS. In Figure 5(a), victim activity A2 intends to start a legitimate utility activity U to serve the user's request

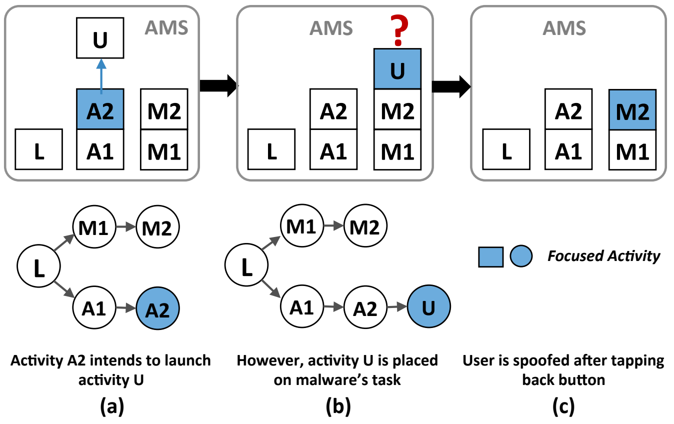
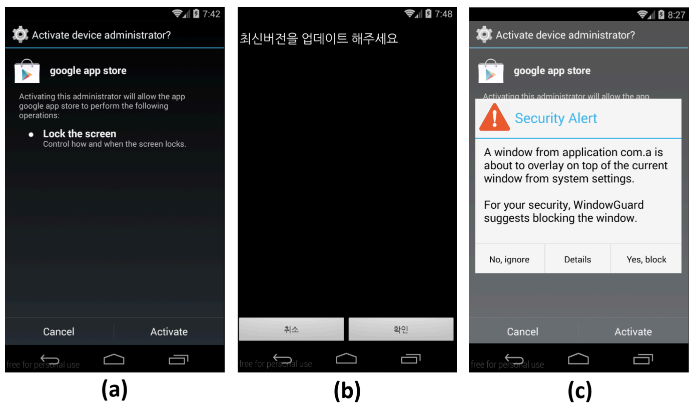

AMS
U
A2
M2
L
A1
M1
?
M1
M2
L
A1
A2
U
Focused Activity
Activity A2 intends to launch activity U
(a)
However, activity U is placed on malware's task
(b)
User is spoofed after tapping back button
(c)

Fig. 5. System state and activity session in a back hijacking attack. A: victim app; M: malware; U: legitimate utility app.



Activate device administrator?

google app store

Activating this administrator will allow the app google app store to perform the following operations:

- **Lock the screen**
  Control how and when the screen locks.

최신버전을 업데이트 해주세요

Security Alert

A window from application com.a is about to overlay on top of the current window from system settings.

For your security, WindowGuard suggests blocking the window.

Fig. 6. Screenshot of (a) Admin privilege confirmation window; (b) tapjacking attack window placed on top of (a) (message: "please update the app to the latest version"); (c) Security alert dialog created by WindowGuard.

(e.g., playing a video). However, when activity U is created, it is tricked to be emplaced on top of a phishing malware's task, whose enclosing activities M1 and M2 are camouflaged as the appearance of A1 and A2. This happens due to the use of *NEW_TASK* intent flag when starting activity U and the abuse of *taskAffinity* by the malware. When the user later taps the back button, the phishing activity M2 is resumed by the system, while the user still believes he/she has gone back to the original activity A2. User's sensitive information (such as bank login information) is then stolen by the malware.

WindowGuard can readily detect the task hijacking behavior when activity U is launched on the malware task. As shown in Figure 5(b), the activity session keeps track of the visible activity windows from the launcher activity all the way to activity U. When activity U is started, the legitimacy of back stacks is checked by comparing related back stacks (i.e. launcher task, task A and task M) against the focused activity session. The disparity of the comparison is obvious due to the relocation of activity U. In this case, a notification is created in the status bar to notify the user of the possible security threat. Likewise, all other task hijacking attacks violating the legitimacy of back stack can be defeated by WindowGuard as well. On the other hand, this kind of task manipulation behavior can also be used for legitimate purpose. WindowGuard provides detailed information about the security risk to the user, and it is up to the user to make the final judgment based on the context. For example, a useful task management app may frequently use various task-related features and inadvertently trigger the alarm. In this case, the user can easily cease the surveillance of this particular app in the WindowGuard control panel. In some other context, the security alert is particularly alarming when the user is working in a bank account, e.g., making a money transfer.

Tapjacking: Clickjacking attack is well known in web security. The similar attack idea is brought to the Android environment, called tapjacking. Tapjacking overlays windows on top of screen and spoofs the user to perform undesirable operations. Bankbot [23] is a family of banking malware discovered in South Korea Android market in October 2014. It was designed to steal authentication information from the clients of various financial institutions. To avoid itself from being uninstalled, Bankbot disguises itself as Google play store app and attempts to acquire admin privileges of the device. Granting admin privilege requires user confirmation, as shown in Figure 6(a). Bankbot overlays a free window (system alert window) on top of the admin privilege confirmation activity, claiming that the software needs update to the latest version. Although the free window is opaque, it is intentionally configured to not receive user tap input, such that the tapping of the button at the bottom of the free window is in turn received by the active window underneath it, i.e., the system confirmation window. As a result, the user agrees to the software update request without realizing that he/she is in fact granting the admin privilege to the malware. WindowGuard immediately detects the attempt of free window launch and pauses it before asking for the user's decision from a security alert dialog, as shown in Figure 6(c). The attack is detected because the current display owner is the system settings, who owns the focused confirmation activity. Any window operations (including free and activity window) that affect the focused activity window are reported to the user and ask for user permission to proceed.

Ransomware: Screen-lock ransomware blackmails victim users by locking the screen for money in exchange for the accessibility to the system again. Ransomware has migrated to Android and has been growing at an alarming rate in the past few years. A ransomware usually renders a high-z-order free window to overlay the full screen and hence blocks all user inputs to the system, leaving the system effectively "locked up". The ransomware can even use a combined GUI attack vectors, e.g., getting admin privilege via tapjacking, to become more powerful and hard to remove, like a recent ransomware called Lockerpin [26]. In addition, [28] demonstrates that a ransomware can also launch activity windows to prohibit user access to targeted victim apps, e.g., an anti-virus app. In either experiment, WindowGuard can block the lock screen window as long as the window's initiator is not the current display owner. Even if the user is spoofed and accidentally gets trapped by a lock screen, the user can always escape by clicking the home or recent button, which starts the launcher or system UI activities. Changing the display owner to launcher or system UI make the foreground malicious lock screen no longer valid and the lock screen is immediately removed. A notification message is then created to inform the user about the security enforcement just occurred.

Security Feature	Alert Msg	# of Apps	% of Apps
Activity Session Legitimacy	T, N	12	0.10
New Window Access Control	D	39	0.32
Existing Window Legitimacy	T, N	14	0.12
New Activity Control	D	69	0.57
Activity Resume Legitimacy	D	11	0.09
Any Feature(s)		124	1.03

TABLE II. NUMBER AND PERCENTAGE OF LEGITIMATE APPS THAT TRIGGER DIFFERENT SECURITY FEATURES OF WINDOWGUARD. TOTALLY 12,060 MOST POPULAR APPS FROM GOOGLE PLAY. ALERT MESSAGES ARE IN FORMS OF - T: TOAST MESSAGE, N: SYSTEM NOTIFICATION, D: CONFIRMATION DIALOG.

# of security features triggered	*1*	*2*	*3*	*4*	*5*
# of Apps	107	15	0	2	0

TABLE III. THE NUMBER OF APPS THAT TRIGGER DIFFERENT NUMBERS OF SECURITY FEATURES.

B. *Usability and Performance Impact*

While providing comprehensive protection of the GUI system, WindowGuard is designed to maximally preserve user experience. In this section, we seek to understand the usability impact of WindowGuard on the legitimate apps. To this end, we collect 12,060 most popular apps from Google play, each app with over 1 million installs. The experiment is conducted on Nexus 5 devices with Android 4.4, the most distributed Android version. To emulate user input, we employ Monkey, a stress testing tool, to generate pseudo-random streams of user events to exercise each app continuously for 5 minutes. In order to take into account of app interactions, at least 30 other apps are installed at the same time when an app is under test. The protection of WindowGuard is classified into 5 security features and each feature maintains a white list. Once an app is white-listed by the user, it is exempt from the security check of the corresponding feature. For example, if app A is added to the white list of "Free Window Control", app A can launch free windows anytime without alerting the user or being blocked by WindowGuard. Once a false alarm is raised, we assume the app is immediately added to the white list of the corresponding security feature by the user (either manually or automatically after user consent). To measure how "annoying" the security features affect the normal function, Table II reports the percentage of apps that trigger each type of security alert in our experiment.

As shown in Table II, WindowGuard has no impact on most popular apps (98.97% apps). It indicates that most of these popular apps follow the Android app model and the norm of app navigation. Among these 1.03% of apps that trigger WindowGuard's alerts, most apps only trigger one security feature, as shown in Table III. It means that WindowGuard only interrupts the user once during the use of most of these apps. It is noteworthy that the "New Window Access Control" and "New Activity Control" features affect 0.32% and 0.57% of all apps respectively. We find that these apps launch free or activity windows on top of other apps for a variety of legit purposes. For example, toast messages are raised by background app services to display warning messages; A free window is decorated as a handy and always-visible controller for a music player app; Certain ad libs create free window or new activities to display advertisements; and app locker apps use free window or activity window to "lock" particular apps before the correct password is provided. Although these windows are all for legitimate purposes, the windows they launched have in fact interfered with the GUI of other apps, even with no bad intention. It is therefore a good time to let the user make the decision on if they are useful or not, e.g., a floating music player controller window is useful but an advertisement window might be annoying for the user. In our experiment, we also find that a significant portion of apps re-write the back button behavior. Instead of "going back", the back button is either disabled or re-directs the user to another activity of the same app. As discussed in Section III-C, WindowGuard considers back behavior modification less risky and does not raise alarm when the back behavior fails to meet user expectation. Despite that, changing the behavior outrageously can be confusing to users and requires careful consideration in app design. Another finding is that, among the apps that trigger security alerts, Table IV shows the number of apps that trigger a particular aggregate number of security alerts in the given testing period. As shown in the figure, a majority of apps triggers less than 3 security alerts even if the app runner is a pseudo-random event generator.

# of security alerts	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*>= 10*
# of Apps	34	28	14	5	6	1	4	0	4	18

TABLE IV. THE NUMBER OF APPS THAT TRIGGER DIFFERENT NUMBERS OF SECURITY FEATURES.

We now proceed to the performance evaluation of WindowGuard. To evaluate the performance overhead, we use Monkey to generate the same sequence of 5000 user events to the same app running on the system with and without WindowGuard module enabled. We collect the complete time T_{with} and $T_{w/o}$, respectively, and the overhead is calculated as $(T_{with} - T_{w/o})/T_{w/o} \times 100\%$. We manually select 100 apps that are in different categories and have complicated activity/window hierarchies. WindowGuard turns out to perform very efficiently, yielding only 0.45% of performance overhead on average.

VI. RELATED WORK

GUI security has been well studied in traditional desktop environments [3], [8], [14], [32]. On the other hand, the unique mobile environment has raised unique challenges.

GUI confidentiality attacks and defenses. Previous research proves that the confidentiality of GUI can be broken through side channels such as shared-memory side channel [7], peeking sensor information [21], [36], via system or app flaws [17], [18], [29], or shoulder surfing [22]. Sensitive GUI information can also be disclosed by taking screen shots because of adb flaws [17], or via embedded malicious UIs [18], [29]. One the other hand, GUI information disclosure can also be put into good use for forensics analysis [30], [31]. A few approaches have been proposed to protect GUI confidentiality [6], [24], [29], which help limit the attack surfaces for confidentiality breaches. However, comprehensively protecting GUI confidentiality from all aspects of the system remains to be an open question. Our work instead focuses on the integrity and availability of Android GUI, properties that are seriously threatened by emerging GUI attacks.

GUI integrity and availability attacks and defenses. Previous research shows the possibility to launch phishing [2], [7], [13] or tapjacking attacks [19], [25] in Android by overlaying a window of attacker's control on top of the victim app's window. It is also viable to manipulate the activity browsing history to launch a variety of task hijacking attack [28]. Denial of service attacks [5], [28] and adware [11], [27], [33] are also posing increasing threat to the GUI availability. Roesner et. al. [29] systematically study the design of secure embedded user interfaces. Bianchi et. al. [2] propose a novel two-layer defense towards defending against GUI confusion attack, an important type of GUI attack. Compared with previous work, we propose a new security model to systematically protect the integrity and availability of the GUI system, while preserving the original user experience. The implementation, WindowGuard, can defeat broader GUI attacks and is practical to be distributed to a large number of Android devices.

Integrity of program execution. Control flow integrity [20] defends against subverted machine-code execution such as return-oriented programming [10] and return-to-libc [15] attacks. One of the approaches is to save the state of the program (e.g. the native return address) in a shadow stack [16], [29], [34]. When the program state is resumed (e.g. function return), the resumed program state is compared with the saved copy on the shadow stack. Similar idea is applied to the legitimacy check of activity session in our work. The previous-visited GUI states, activities, are saved in a activity session. To defeat task hijacking attacks, the integrity of the foreground activity session is scrutinized whenever an activity obtains focus.

VII. Discussion

WindowGuard is not a malware detection system. The goal of WindowGuard is to accurately detect the attacks that affect the GUI integrity and availability of other apps, instead of detecting malicious behavior within an app itself, e.g., a phishing activity within the malware's context. WindowGuard always respects the user's choice. Therefore, if a malware is intentionally launched by the user (e.g., the user is spoofed by the social engineering tricks used by the malware) WindowGuard does not disagree with user's decision. Defending against trojan horse malware like this is out of scope of this defense mechanism. On the other hand, WindowGuard guarantees to prevent a malware from becoming the display owner if the user or the current display owner app does not explicitly launch the malware, as we have seen in our evaluation. In addition, WindowGuard is not a vulnerability discovery system. It is not designed to discover or address the GUI security issues within an app itself, e.g. misleading app navigation design, or vulnerable access control of an app component, although WindowGuard does has the capability to detect a subset of these design flaws and give hints to the user, such as inconsistency of back button behavior.

There are several limitations of WindowGuard. First, although WindowGuard can successfully detect all known GUI attacks, it also introduces false positives. As we have seen in evaluation, legitimate app developers, without understanding the security implications, may conduct operations that violate AWI principles. For instance, a phone call recorder app namely FonTel displays a window (which contains voice recording control buttons) on top of the system dialer app whenever there is a phone call. Although the window is useful for the apps functionality, it has effectively disturbed another app's GUI and user experience. Determining the real intention of such app behavior (e.g. an useful phone recorder control widget or a phishing window) is fundamentally difficult for automatic systems. In contrast, users are more capable of making the best decision based on the runtime context. WindowGuard adopts the advantage of user to overcome this difficulty while still retaining the original user experience. Second, user involvement may adversely introduces false negative caused by user mistakes, e.g., an user explicitly allows a login pop-up window, which is in fact a phishing window. In this paper, although a security warning is displayed (e.g., in Figure 6), we specify WindowGuard to always respect user decision and report the usibility findings in Section V-B. How to improve security-and-usability balance and the efficacy of defense requires comprehensive user study (e.g., the user study of Android permission system [12]) and is beyond the scope of this paper. Lastly, the implementation of WindowGuard is based on Xposed, which can only be used on rooted Android devices.

VIII. Conclusion

In conclusion, we propose a new security model - Android Window Integrity - to systematically protect Android GUI system from attacks that compromise GUI integrity and availability. We develop WindowGuard, an Xposed module that implements AWI model while preserving the original Android user experience. Our evaluation shows that WindowGuard can successfully defeat all known GUI attacks and yields small impact on usability and performance.

IX. Acknowledgment

We would like to thank anonymous reviewers whose comments help us improve the quality of this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CCF-1320605, NSF SBE-1422215 and CNS-1618684.

References

[1] "Task and Back Stack," http://developer.android.com/guide/components/tasks-and-back-stack.html.

[2] A. Bianchi and J. Corbetta and L. Invernizzi and Y. Fratantonio and C. Kruegel and G. Viana, "What the App is That? Deception and Countermeasures in the Android User Interface," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2015.

[3] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking Revisited: A Perceptual View of UI Security," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[4] "Police Locker land on Android Devices," 2014, http://malware.dontneedcoffee.com/2014/05/police-locker-available-for-your.html.

[5] "Simplocker: First Confirmed File-Encrypting Ransomware for Android," 2014, http://www.symantec.com/connect/blogs/simplocker-first-confirmed.

[6] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan, "You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps," in *Proceedings of the USENIX Security Symposium*, 2015.

[7] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks," in *Proceedings of the USENIX Security Symposium*, 2014.

[8] S. Chen, J. Meseguer, R. Sasse, H. wang, and Y. Wang, "A Systematic Approach to Uncover Security FLaws in GUI Logic," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2007.

[9] D. Dinkar, P. Greve, K. Landfield, F. Raget, and E. Peterson, "McAfee Labs Threats Report," Intel Security, Tech. Rep., 2016.

[10] E. Buchanan and R. Roemer and H. Shacham and S. Savage, "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] E. Erturk, "A case study in open source software security and privacy: Android adware," in *World Congress on Internet Security (WorldCIS)*, 2012.

[12] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "AndroidPermissions: User Attention, Comprehension and Behavior," in *Proceedings of Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[13] A. P. Felt and D. Wagner, "Phishing on Mobile Devices," in *Web 2.0 Security and Privacy*, 2011.

[14] N. Feske and C. Helmuth, "A Nitpickers Guide to a Minimal-complexity Secure GUI," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2005.

[15] H. Shacham, "The Geometry of Innocent Flesh on the Boan: Return-into-libc without function calls (on the x86)," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2007.

[16] L. Davi and A. Sadeghi and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks," in *Proceedings of ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011.

[17] C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to Milk Your Android Screen for Secrets," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2014.

[18] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android System," in *Proceedings of Annual Computer Security Applications Conference*, 2011.

[19] T. Luo, X. Jin, and A. Ananthanarayanan, "Touchjacking Attacks on Web in Android, iOS, and Windows Phone," in *Proceedings of the 5th international conference on Foundations and Practice of Security (FPS)*, 2012.

[20] M. Abadi and M. Budiu and U. Erlingsson and J. Ligatti, "Control Flow Integrity," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2005.

[21] E. Miluzzo, A. Varshavsky, and S. Balakrishnan, "TapPrints: Your Finger Taps Have Fingerprints," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[22] M. Mitchell, A. Wang, and P. Reiher, "Cashtags: Prevent Leaking Sensitive Information through Screen Display," in *Proceedings of the USENIX Security Symposium*, 2015.

[23] "Mobile Threats in October 2014," 2014, https://news.drweb.com/show/?i=7061&lng=en.

[24] N. Zhang and K. Yuan and M. Naveed and X. Zhou and X. Wang, "Leave Me Alone: App-level Protection Against Runtime Information Gathering on Android," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2015.

[25] M. Niemietz and J. Schwenk, "UI Redressing Attacks on Android Devices," in *Blackhat*, 2012.

[26] "Aggressive Android ransomware spreading in the USA," 2015, http://www.welivesecurity.com/2015/09/10/aggressive-android-ransomware-spreading-in-the-usa/.

[27] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are These Ads Safe: Detecting Hidden Attacks Through the Mobile App-Web Interfaces," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[28] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android," in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2015.

[29] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *Proceedings of the USENIX Security Symposium*, 2013.

[30] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing Together Android App GUIs from Memory Images," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[31] B. Saltaformaggio, R.Bhatia, Z. Gu, X. Zhang, and D. Xu, "VCR: App-Agnostic Recovery of Photographic Evidence from Android Device Memory Images," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2015.

[32] J. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, "Design of the EROS Trusted Window System," in *Proceedings of the USENIX Security Symposium*, 2004.

[33] S. Son, D. Kim, and V. Shatikov, "What Mobile Ads Know About Mobile Users," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[34] "A Stack Smashing technique Protection Tool for Linux," http://www.angelfire.com/sk/stackshield/.

[35] "The Android Trojan Svpeng Now Capable of Mobile Phishing," 2013, https://securelist.com/blog/research/57301/the-android-trojan-svpeng-now-capable-of-mobile-phishing/.

[36] Z. Xu, K. Bai, and S. Zhu, "TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Netowrks (WiSec)*, 2012.