

Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks

Qi Alfred Chen, Zhiyun Qian[†], Z. Morley Mao
University of Michigan, [†]University of California, Riverside
alfchen@umich.edu, zhiyunq@cs.ucr.edu, zmao@umich.edu

Abstract

The security of smartphone GUI frameworks remains an important yet under-scrutinized topic. In this paper, we report that on the Android system (and likely other OSes), a weaker form of GUI confidentiality can be breached in the form of UI state (not the pixels) by a background app without requiring any permissions. Our finding leads to a class of attacks which we name *UI state inference attack*. The underlying problem is that popular GUI frameworks by design can potentially reveal every UI state change through a newly-discovered public side channel — shared memory. In our evaluation, we show that for 6 out of 7 popular Android apps, the UI state inference accuracies are 80–90% for the first candidate UI states, and over 93% for the top 3 candidates.

Even though the UI state does not reveal the exact pixels, we show that it can serve as a powerful building block to enable more serious attacks. To demonstrate this, we design and fully implement several new attacks based on the UI state inference attack, including hijacking the UI state to steal sensitive user input (*e.g.*, login credentials) and obtain sensitive camera images shot by the user (*e.g.*, personal check photos for banking apps). We also discuss non-trivial challenges in eliminating the identified side channel, and suggest more secure alternative system designs.

1 Introduction

The confidentiality and integrity of applications' GUI content are well recognized to be critical in achieving end-to-end security [1–4]. For instance, in the desktop and browser environment, window/UI spoofing (*e.g.*, fake password dialogs) breaks GUI integrity [3,4]. On the Android platform, malware that takes screenshots breaches GUI confidentiality [5]. Such security issues can typically lead to the direct compromise of the confidentiality of user input (*e.g.*, keystrokes). However, a weaker form of confidentiality breach has not been thoroughly explored, namely the knowledge of the application UI state (*e.g.*, knowing that the application is showing a login window) *without knowing the exact pixels of the screen*, especially in a smartphone environment.

Surprisingly, in this paper we report that on the Android system (and likely on other OSes), such GUI confidentiality breach is indeed possible, leading to serious security consequences. Specifically, we show that *UI state* can be inferred without requiring any Android permissions. Here, *UI state* is defined as a mostly consistent user interface shown in the window level, reflecting a specific piece of program functionality. An example of a UI state is a login window, in which the text content may change but the overall layout and functionality remain the same. Thus, we call our attack *UI state inference attack*. In this attack, an attacker first builds a UI state machine based on UI state signatures constructed offline, and then infers UI states in real time from an unprivileged background app. In Android terminology, the UI state is known as *Activity*, so we also call it *Activity inference attack* in this paper.

Although UI state knowledge does not directly reveal user input, due to a lack of direct access to the exact pixels or screenshots, we find that it can effectively serve as a building block and enable more serious attacks such as stealing sensitive user input. For example, based on the inferred UI states, we can further break the GUI integrity by carefully exploiting the designed functionality that allows UI preemption, which is commonly used by alarm or reminder apps on Android.

The fundamental reason for such confidentiality breach is in the Android GUI framework design, where every UI state change can be unexpectedly observed through publicly accessible side channels. Specifically, the major enabling factor is a newly-discovered *shared-memory side channel*, which can be used to detect window events in the target application. This side channel exists because shared memory is commonly adopted by window managers to efficiently receive window changes or updates from running applications. For more details, please refer to §2.1 where we summarize the design and implementation of common window managers, and §3.2 where we describe how shared memory plays a critical role. In fact, this design is not specific to Android: nearly all popular OSes such as Mac OS X, iOS, and Windows also adopt this shared-memory mechanism for their win-

indow managers. Thus, we believe that our attack on Android is likely to be generalizable to other platforms.

Since the window manager property we exploit has no obvious vulnerabilities in either design or implementation, it is non-trivial to construct defense solutions. In §9, we discuss ways to eliminate the identified side channels, and also suggest more secure alternative system designs.

Our discovered Activity inference attack enables a number of serious follow-up attacks including (1) *Activity hijacking attack* that can unnoticeably hijack the UI state to steal sensitive user input (e.g., login credentials), and (2) *camera peeking attack* that captures sensitive camera images shot by the user (e.g., personal check photos for banking apps). We have fully designed and implemented these attacks and strongly encourage readers to view several short video demos at <https://sites.google.com/site/uistateinferenceattack/demos> [6].

Furthermore, we demonstrate other less severe but also interesting security consequences: (1) existing attacks [5, 7–10] can be enhanced in stealthiness and effectiveness by providing the target UI states; (2) user behavior can be inferred through tracking UI state changes. Previous work has demonstrated other interesting Android side-channel attacks, such as inferring the web pages a user visits [11] as well as the identity, location, and disease information of users [12]. However, these attacks are mostly application-specific with limited scope. For instance, Memento [11] only applies to web browsers, and Zhou *et al.* [12] reported three side channels specific to three different apps. In contrast, the UI state concept in this paper applies generally to all Android apps, leading to not only a larger attack coverage but also many more serious attacks, such as the Activity hijacking attack which violates GUI integrity.

The contributions of this paper are as follows:

- We formulate the general UI state inference attack that violates a weaker form of GUI confidentiality, aimed at exposing the running UI states of an application. It exploits the unexpected interaction between the design and implementation of the GUI framework (mainly the window manager) and a newly-discovered shared-memory side channel.
- We design and implement the Android version of this attack and find an accuracy of 80–90% in determining the foreground Activity for 6 out of 7 popular apps. The inference itself does not require any permissions.
- We develop several attack scenarios using the UI state inference technique and demonstrate that an attacker can steal sensitive user input and sensitive camera images shot by the user when using Android apps.

For the rest of the paper, we first provide the attack background and overview in §2. The newly-discovered side channel and Activity transition detection are detailed

in §3, and based on that, the Activity inference technique is described in §4. In §5, we evaluate this attack with popular apps, and §6, §7 and §8 show concrete follow-up attacks. We cover defense in §9, followed by related work in §10, before concluding in §11.

2 Background and Overview

2.1 Background: Window Manager

Window manager is system software that interacts with applications to draw the final pixels from all application windows to the frame buffer, which is then displayed on screen. After evolving for decades, the most recent design is called *compositing window manager*, which is used virtually in all modern OSes. Unlike its predecessors, which allow individual applications to draw to the frame buffer directly, a compositing window manager requires applications to draw the window content to *off-screen buffers* first, and use a dedicated window compositor process to combine them into a final image, which is then drawn to the frame buffer.

Client-drawn and server-drawn buffer design. There are two types of compositing window manager design, as shown in Fig. 1. In this figure, *client* and *server* refer to the application and the window compositor¹ respectively. In the client-drawn buffer design, the applications draw window content to off-screen buffers, and use IPC to communicate these buffers with the server, where the final image is composited and written to the frame buffer. This design is very popular and is used in Mac OS X, iOS, Windows, Android, and Wayland for the future Linux [13, 14]. In the server-drawn buffer design, the main difference is that the off-screen buffers are allocated and drawn by the window compositor instead of by the applications. Applications send commands to the window compositor to direct the drawing process. Only the X window system on the traditional Linux and Mir for the future Linux [15] use this design.

Both designs have their advantages. The client-drawn buffer design provides better isolation between applications, more flexible window drawing and more balanced overhead between the client and the server. For the server-drawn buffer design, the server has control over all applications’ window buffers, which is better for centralized resource management. Interestingly, some prior work choose the former to enhance GUI security [1], but we find that it actually enables our attacks (shown in §3).

2.2 Background: Android Activity and Activity Transition

In Android, the UI state our attack infers is called *Activity*. An Activity provides a user interface (UI) for user in-

¹For traditional Linux the server is an X server, and the window compositor is a separate process talking to the X server. In Fig. 1 we refer to the combination of them as the window compositor.

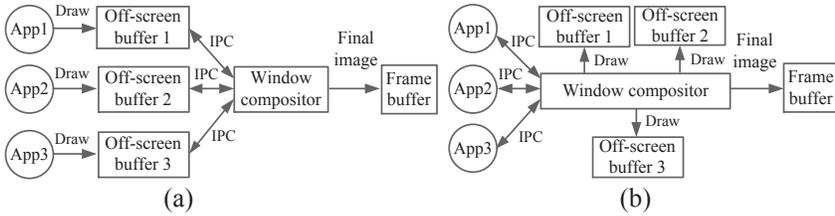


Figure 1: Two types of compositing window manager design: (a) client-drawn buffer design, and (b) server-drawn buffer design. *Client* refers to the application, and *server* refers to the window compositor.

teractions, and is typically a full-screen window serving as a functionality unit in Android. We denote Activities as a, b, \dots , and the set of Activities for an app as A . Due to security concerns, by default apps cannot know which Activity is currently shown in the foreground unless they are the owners or the central Activity manager.

An Activity may display different content depending on the app state. For instance, a dictionary app may have a single “definition” Activity showing different texts for each word lookup. We call these distinct displays *ViewStates*, and denote the set of them for Activity a as $a.VS$. **Activity transition.** In Android, multiple Activities typically work together and transition from one to another to support the functionality of an app as a whole. An example is shown in Fig. 2. During a typical transition, the current foreground Activity pauses and a new one is created. A Back Stack [16] storing the current and past Activities is maintained by Android. To prevent excessive memory usage, at any point in time, only the top Activity has its window buffer allocated. Whenever an Activity transition occurs, the off-screen buffer allocation for the new Activity window and the deallocation for the existing Activity window take place.

Activity transitions can occur in two ways: a new Activity is created (*create* transition), or an existing one resumes when the BACK key is pressed (*resume* transition), corresponding to push and pop actions in the Back Stack. Fig. 3 shows the major function calls involved in these two transition types. Both transition types start by pausing the current foreground Activity, and then launching the new one. During launch, the *create* transition calls both `onCreate()` and `onResume()`, while the *resume* transition only calls `onResume()`. Both `onCreate()` and `onResume()` are implemented by the app. After that, `performTraversal()` is called, in which `measure()` and `layout()` calculate the sizes and locations of UI components, and `draw()` puts them into a data structure as the new Activity UI. Finally, the *create* transition pushes the new Activity into the Back Stack and stops the current one, while the *resume* transition pops the current one and destroys it.

Activity transition graph. Immediately after a tran-

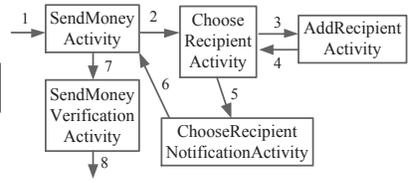


Figure 2: Activities involved in sending money in Android Chase app. The numbers denote the action order.

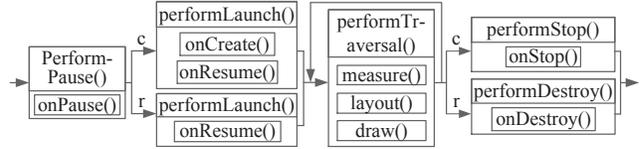


Figure 3: The function call trace for *create* (denoted by c) and *resume* (denoted by r) transitions.

sition, the user lands on one of the ViewStates of the new Activity, which we call a *LandingState*. We denote the set of LandingStates for Activity a as $a.LS$, and $a.LS \subseteq a.VS$. Individual LandingStates are denoted as $a.ls_1, a.ls_2, \dots$. Activity transition is a relationship $a.VS \rightarrow b.LS, a, b \in A$. As the ViewState before the transition is not of interest in this study, we simplify it to $a \rightarrow b.LS$, which forms the graph in Fig. 4. Note that our definition is slightly different from that in previous work [17] as the edge tails in our graph are more fine-grained: they are LandingStates instead of Activities.

2.3 Attack Overview

Our proposed UI state inference is a general side-channel attack against GUI systems, aimed at exposing the running UI state of an application at the window level, *i.e.*, the currently displayed window (without knowing the exact pixels). To achieve that, the attack exploits a newly-discovered shared-memory side channel, which may exist in nearly all popular GUI systems used today (shown in §3). In this paper, we focus on the attack on the Android platform: Activity inference attack. We expect the technique to be generalizable to all GUI systems with the same window manger design as that in Android, such as the GUI systems in Mac OS X, iOS, Windows, *etc.*

Threat model. We require an attack app running in the background on the victim device, which is a common requirement for Android-based attacks [7–11, 18]. To ensure stealthiness, the app should be low-overhead, not draining the battery too quickly. Also, as the purpose of permissions is to alert users to privacy- or security-invasive apps [19], the attack should not require any additional permissions besides a few very common ones, for example the INTERNET permission.

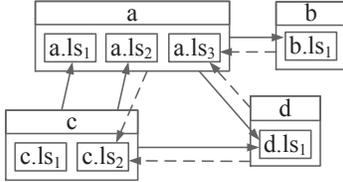


Figure 4: An example Activity transition graph. Solid and dotted edges denote *create* and *resume* transitions respectively.

General steps. As shown in Fig. 5, Activity inference is performed in two steps:

1. Activity transition detection: we first detect an Activity transition event, which reports a single bit of information on whether an Activity transition just occurred. This is enabled by the newly-discovered shared-memory side channel. As shown later in §3.3, the change observed through this channel is a highly-distinct “signal”.

2. Activity inference: upon detecting an Activity transition, we need to differentiate which Activity is entering the foreground. To do so, we design techniques to train the “signature” for the landing Activity, which roughly characterizes its starting behavior through publicly observable channels, including the new shared-memory side channel, CPU utilization time, and network activity (described in §4).

Finally, using our knowledge of the foreground Activity in real time, we develop novel attacks that can effectively steal sensitive user input as well as other information as detailed in §6, §7 and §8.

3 Shared-Memory Side Channel and Activity Transition Detection

In this section, we first report the newly-discovered side channel along with the fundamental reason for its existence, and then detail the transition detection technique.

3.1 Shared-Memory Side Channels

As with any modern OS design, the memory space of an Android app process consists of the private space and the shared space. Table 1 lists memory counters in `/proc/pid/statm` and their names used in the Linux command `top` and the Linux source code. Inherited from Linux, these counters can be freely accessed without any privileges. With these counters, we can calculate both the private and shared sizes for virtual memory and physical memory. In this paper, we leverage `mm->shared_vm` and `file_rss` as our shared-memory side channels, the former for virtual memory and the latter for physical memory. For convenience, we refer to them as *shared_vm* and *shared_pm*. In this section, we focus on using *shared_vm* to detect Activity transition events. In

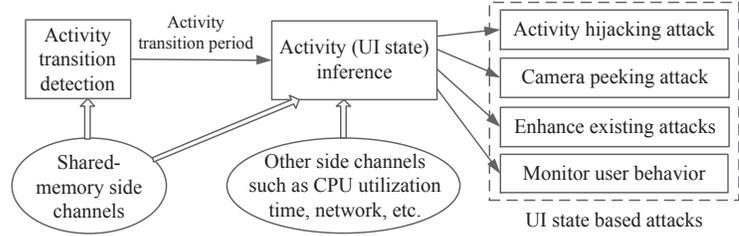


Figure 5: Activity inference attack overview.

§4.1, we use both *shared_vm* and *shared_pm* to infer Android Content Provider usages in the Activity inference, which is another use case we discovered.

3.2 Android Window Events and Shared-Memory Side Channel

We find that *shared_vm* changes are correlated with Android window events. In this section, we detail its root cause and prevalence in popular GUI systems.

Shared-memory IPC used in the Android window manager. As mentioned earlier in §2.1, Android adopts the client-drawn buffer design, where each client (app) needs to use a selected IPC mechanism to communicate their off-screen buffers with the window compositor. In practice, we find that shared memory is often used, since it is the most efficient channel (without large memory copy overhead). On Android, when an Activity transition occurs, *shared_vm* size changes can be found in both the app process and the window compositor process named *SurfaceFlinger*. More investigations into Android source code reveal that the size changes correspond to the allocations and deallocations of a data structure named *GraphicBuffer*, which is the off-screen buffer in Android. In the Android window drawing process shown in Fig. 3, *GraphicBuffer* is shared between the app process and the *SurfaceFlinger* process using `mmap()` at the beginning of `draw()` in `performTraversal()`.

Interestingly, this implies that if we know the *GraphicBuffer* size for a target window, we can detect its allocation and deallocation events by monitoring the size changes of *shared_vm*. Since the *GraphicBuffer* size is proportional to the window size, and an Activity is a full-screen window, its *GraphicBuffer* size is fixed for a given device, which can be known beforehand.

It is noteworthy that different from private memory space, shared memory space changes only when shared files or libraries are mapped into the virtual memory. This keeps our side channel clean; as a result, the changes in *shared_vm* are distinct with minimum noise.

Shared-memory side-channel vulnerability on other OSes. To understand the scope, we investigate other OSes besides Android. On Linux, Wayland makes it clear that it uses shared buffers as IPC between the win-

Item in <code>/proc/pid/statm</code>	Description	Name in <code>top</code>	Name in Linux source code
<code>VmSize</code>	Total virtual memory size	VIRT	<code>mm->total_vm</code>
<code>drs</code>	Private virtual memory size	/	<code>mm->total_vm - mm->shared_vm</code>
<code>resident</code>	Total physical memory size	RES	<code>file_rss+anon_rss</code>
<code>share</code>	Shared physical memory size	SHR	<code>file_rss</code>

Table 1: Android/Linux memory counters in `/proc/pid/statm` and their names in the Linux command `top`, and the Linux source code (obtained from `task_statm()` in `task_mmu.c`). The type of `mm` is `mm_struct`.

down compositor and clients to render the windows [13]. Similar to Android, attackers can use `/proc/pid/statm` to get the shared memory size and detect window events.

Mac OS X, iOS and Windows neither explain this IPC in their documentations nor have corresponding source code for us to explore, so we did some reverse engineering using memory analysis tools such as VMMAP [20]. On Windows 7, we found that whenever we open and close a window, a memory block appears and disappears in the shared virtual memory space of both the window compositor process, *Desktop Window Manager*, and the application process. Moreover, the size of this memory block is proportional to the window size on the screen. This strongly implies that this memory block is the off-screen buffer and shared memory is the IPC used for passing it to the window compositor. Thus, using the `GetProcessMemoryInfo()` API that does not require privilege, similar inference may be possible.

Mac OS X is similar to Windows except that the memory block in shared memory space is named *CG backing store*. On iOS this should be the same as Mac OS X since they share the same window compositor, *Quartz Compositor*. But on Mac OS X and iOS, only system-wide aggregated memory statistics seem available through `host_statistics()` API, which may still be usable for this attack but with a less accuracy.

3.3 Activity Transition Detection

With the above knowledge, detecting Activity transition is simply a matter of detecting the corresponding window event pattern by monitoring `shared_vm` size changes.

The left half of Fig. 6 shows the typical `shared_vm` changing pattern for an Activity transition, and we name it *Activity transition signal*. In this signal, the positive and negative spikes are increases and decreases in `shared_vm` respectively, corresponding to `GraphicBuffer` allocations and deallocations. The `GraphicBuffer` allocation for the new Activity usually happens before the deallocation for the current Activity, which avoids user-visible UI delays. Since Activity windows all have full-screen sizes, the increase and decrease amount are the same. With the multiple buffer mechanism for UI drawing acceleration on Android [21], 1–3 `GraphicBuffer` allocations or deallocations can be observed during a sin-

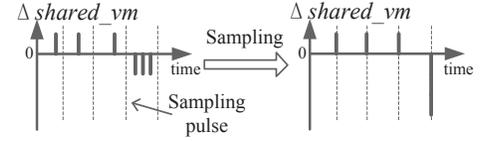


Figure 6: A successful sampling of an Activity transition signal for the Activity transition detection

gle transition, resulting in multiple spikes in Fig. 6. The delay between allocations is usually 100–500 ms due to measurement and layout computations, while the delay between deallocations is usually under 10 ms. An example result of a successful sampling is shown on the right half of Fig. 6 with the sampling period being 30–100 ms.

To detect this signal, we monitor the changes of `shared_vm`, and conclude an *Activity transition period* by observing (1) both full-screen size `shared_vm` increase and decrease events, (2) the idle time between two successive events is longer than a threshold `idle_thres`. A successful detection is shown on the top of Fig. 10.

We evaluate this method and find a very low false positive rate, which is mainly because the `shared_vm` channel is clean. In addition, it is rare that the following unique patterns happen randomly in practice: (1) the `shared_vm` increase and decrease amounts are exactly the same as the full-screen `GraphicBuffer` size (920 pages for Samsung Galaxy S3); (2) both the increase and decrease events occur very closely in time.

On the other hand, this method may have false negatives due to a cancellation effect — when an increase and a decrease are in the same sampling period, they cancel each other and the `shared_vm` size stays unchanged. Raising the sampling rate can overcome this problem, but at the cost of increased sampling overhead. Instead, we solve the problem using the number of minor page faults (`minflt`), in `/proc/pid/stat`. When allocating memory for a `GraphicBuffer`, the physical memory is not actually allocated until it is used. At time of use, the same number of pages faults as the allocated `GraphicBuffer` page size is generated. Since `minflt` monotonically increases as a cumulative counter, we can use it to deduce the occurrence of a cancellation event.

4 Activity Inference

After detecting an Activity transition, we infer the identity of the new Activity using two kinds of information:

1. Activity signature. Among functions involved in the transition (as shown in Fig. 3), `onCreate()` and `onResume()` are defined in the landing Activity, and the execution of `performTraversal()` depends on the UI elements and layout in its `LandingState`. Thus, every transition has behavior specific to the landing Activ-

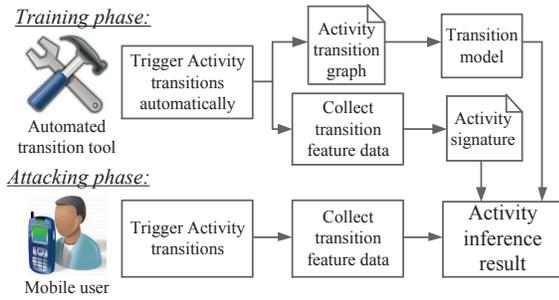


Figure 7: Overview of the Activity inference

ity, giving us opportunities to infer the landing Activity based on feature data collected during the transition.

2. Activity transition graph. If the Activity transition graph of an app is sparse, once the foreground Activity is known, the set of the next candidate Activities is limited, which can ease the inference difficulty. Thus, we also consider Activity transition graph in the inference.

Fig. 7 shows an overview of the Activity inference process. This process has two phases, the training phase and the attacking phase. The training phase is executed first offline. We build a tool to automatically generate Activity transitions in an app, and at the same time collect feature data to build the Activity signature and construct the Activity transition graph. In the attacking phase, when a user triggers an Activity transition event, the attack app first collects feature data like in the training phase, then leverages Activity signature and a transition model based on the Activity transition graph to perform inference.

4.1 Activity Signature Design

During the transition, we identify four types of features described below and use them jointly as the signature.

Input method events. Soft keyboard on smartphones is commonly used to support typing in Activities. It usually pops up automatically at the landing time. There is also a window event for the keyboard process, which again can be inferred through *shared_vm*. This is a binary feature indicating whether the LandingState requires typing.

Content Provider events. Android component Content Provider manages access to a structured set of data using SQLite as backend. To deliver content without memory copy overhead, Android uses anonymous shared memory (*ashmem*) shared by the Content Provider and the app process. Similar to the compositing window manger design, by monitoring *shared_vm*, we can detect the query and release events of the Content Provider. Specifically, in Android design, we found that the virtual memory size of *ashmem* allocated for a Content Provider query is a fixed large value, e.g., 2048 KB for Android 4.1, which creates a clear signal. Usually its content only constitutes a small portion. To know the content size, we also

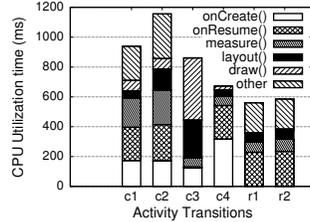


Figure 8: CPU utilization time breakdown for 6 Activity transitions in WebMD

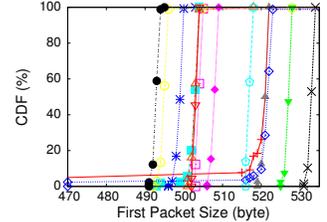


Figure 9: CDF of the first packet sizes for various LandingStates in H&R Block

monitor *shared_pm* introduced in §3.1, which indicates the physical memory allocation size for the content.

The Content Provider is queried in `onCreate()` and `onResume()` to show content in the landing Activity. For signature construction, we collect Content Provider query events and the corresponding content size by monitoring *shared_vm* and *shared_pm*. As *shared_pm* may be noisy, we use a normal distribution to model the size.

CPU utilization time. Fig. 8 shows the CPU utilization time collected by DDMS [22] for each function in Fig. 3 during the transition. For the 6 transitions, *c* and *r* denote *create* and *resume* transition, and 1–4 denote 4 different LandingStates. The time collected may be inflated due to the overhead added by DDMS profiling. The figure shows that CPU utilization time spent in each function differs across distinct LandingStates due to distinct drawing complexity, and for the same LandingState, *resume* transitions usually take less time than *create* ones since the former lacks `onCreate()`. Thus, the CPU utilization time can be used to distinguish Activity transitions with different transition types and LandingStates.

To collect data for this feature, we record the user space CPU utilization time value (*utime*), in `/proc/pid/stat` for the Activity transition. Similar to previous work [23, 24], we find that the value for the same repeated transition roughly follows normal distribution. Thus, we model it using a normal distribution.

Network events. For LandingStates with content from the network, network connection(s) are initiated in `performLaunch()` during the transition. For a given LandingState, the request command string such as HTTP GET is usually hard-coded, with only minor changes from app states or user input. This leads to similar size of the first packet immediately after the connection establishment. We do not use the response packet size, since the result may be dynamically generated. Fig. 9 shows the CDF of the first packet sizes for 14 Activity LandingStates in H&R Block. As shown, most distributions are quite stable, with variations of less than 3 bytes.

To capture the first packet size, we monitor the send packet size value in `/proc/uid_stat/uid/tcp_snd`. We con-

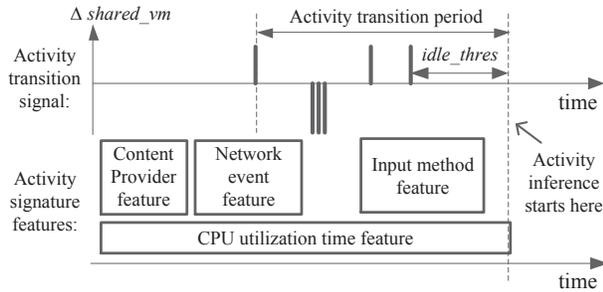


Figure 10: Signature feature data collection timeline.

currently monitor `/proc/net/tcp6`, which contains network connection information such as the destination IP address and app uid for all apps. We use the uid to find the app which the connection belongs to, and use time correlation to match the first packet size to the connection. For the LandingState with multiple connections, we use the *whois* IP address registry database [25] to get the organization names and thus distinguish connections with different purpose. To read first packet sizes accurately, we raise the sampling rate to be 1 in 5 ms during the transition period. Since this period usually lasts less than 1 second, the overall sampling overhead is still low.

For signature construction, we keep separate records for connections with different organization names and occurrence ordering. For each record, we use the first packet size appearance frequencies to calculate the probability for this feature.

Fig. 10 shows the data collection timeline for these feature data and their relationship with the *shared_vm* signal. The Content Provider event feature is collected before the first *shared_vm* increase, and the input method event feature is collected after the first *shared_vm* decrease. Network events are initiated before the first *shared_vm* increase, while the first packet size is usually captured after that. The CPU utilization time feature is collected throughout the whole transition period.

With these four types of features, our signature probability $Prob(\langle \cdot, a.ls_i \rangle)$, $a \in A$, $a.ls_i \in a.LS$ is obtained by computing the product of each feature’s probability, based on the observation that they are fairly independent. In §5, we evaluate our signature design with these four features both jointly and separately.

4.2 Transition Model and Inference Result

Transition model. In our inference, the states (*i.e.*, Activities) are not visible, so we use Hidden Markov Model (HMM) to model Activity transitions. We denote the foreground Activity trace with n Activity transitions as $\{a_0, a_1, \dots, a_n\}$. The HMM can be solved using the Viterbi algorithm [26] with initialization $Prob(\{a_0\}) = \frac{1}{|A|}$, and inductive steps $Prob(\{a_0, \dots, a_n\}) = \arg \max_{a_n.ls_i \in a_n.LS}$

$$Prob(\langle \cdot, a_n.ls_i \rangle) Prob(a_n | a_{n-1}) Prob(\{a_0, \dots, a_{n-1}\}).$$

In inductive steps, $Prob(\langle \cdot, a_n.ls_i \rangle)$ denotes the probability calculated from Activity signature, and $Prob(a_n | a_{n-1})$ denotes the probability that a_{n-1} transitions to a_n . If a_{n-1} has x egress edges in the transition graph, $Prob(a_n | a_{n-1}) = \frac{1}{x}$, assuming that user choices are uniformly distributed.

The typical Viterbi algorithm [26] calculates the most likely Activity trace $\{a_0, a_1, \dots, a_n\}$, with computation complexity $O((n+1)|A|^2)$. However, for our case, only the new foreground Activity a_n is of interest, so we modify the Viterbi algorithm by only calculating $Prob(\{a_{n-c+1}, \dots, a_n\})$, where c is a constant. This reduces the computation complexity to $O(c|A|^2)$. In our implementation, we choose $c = 2$.

Inference result. After inference, our attack outputs a list of Activities in decreasing order of their probabilities.

4.3 Automated Activity Transition Tool

By design, both the Activity signature and Activity transition graph are mostly independent of user behavior; therefore, the training phase does not need any victim participation. Thus, we also develop an automated tool to accelerate the training process offline.

Implementation. Our tool is built on top of `ActivityInstrumentationTestCase`, an Android class for building test cases of an app [27]. The implementation has around 4000 lines of Java code.

Activity transition graph generation with depth-first search. To generate the transition graph, we send and record user input events to Activities to drive the app in a depth-first search (DFS) fashion like the depth-first exploration described in [17]. The DFS graph has ViewStates as nodes, user input event traces as edges (*create* transitions), and the BACK key as the back edge (*resume* transitions). Once the foreground Activity changes, transition information such as the user input trace and the landing Activity name is recorded. The graph generated is in the form of the example shown in Fig. 4.

Activity transition graph traversal. With the transition graph generated, our tool supports automatic graph traversals in deterministic and random modes. In the random mode, the tool chooses random egress edges during the traversal, and goes back with some probabilities.

Tool limitations. We assume Activities are independent from each other. If changes in one Activity affect Activity transition behavior in another, our tool may not be aware of that, leading to missed transition edges. For some user input such as text entering, the input generation is only a heuristic and cannot ensure 100% coverage. To address such limitations, some human effort is involved to ensure that we do not miss the important Activities and ViewStates.

Application name	Activity number	Activity transitions					Activity LandingStates			
		<i>create</i> type	<i>resume</i> type	Total	Graph density	Average egress edge per Activity	Number	w/ content provider	w/ input method	w/ network
WebMD	38	274	129	403	14.0%	10.0	92	18.7%	15.3%	70.3%
Chase	34	257	39	296	17.4%	8.71	50	4%	0.00%	44.0%
Amazon	19	209	190	399	55.3%	21.0	39	0.00%	7.69%	74.3%
NewEgg	55	242	253	495	8.2%	9.0	80	0.00%	8.75%	97.5%
GMail	7	10	10	20	20.4%	2.86	17	0.00%	5.71%	5.8%
H&R Block	20	58	39	97	12.1%	4.85	42	0.00%	2.3%	100%
Hotel.com	24	29	41	70	6.1%	2.92	35	0.00%	2.8%	100%

Table 2: Characteristics of Activity, Activity LandingStates and Activity transitions of selected apps (numbers are obtained manually as the ground truth). The CPU utilization time feature is omitted since it is always available.

Application name	Activity transition detection			Activity inference accuracy		
	Accuracy	FP	FN	Top 1 cand.	Top 2 cand.	Top 3 cand.
WebMD	99%	0.50%	1.0%	84.5%	91.4%	93.6%
Chase	99.5%	0.53%	0.63%	83.1%	91.8%	95.7%
Amazon	99.3%	4%	0.7%	47.6%	65.6%	74.1%
NewEgg	98.4%	0.1%	1.6%	85.9%	92.6%	96.3%
GMail	99.2%	0%	0.8%	92.0%	98.3%	99.3%
H&R Block	97.7%	2%	2.3%	91.9%	96.7%	98.1%
Hotel.com	96.5%	0.6%	3.5%	82.6%	92.7%	96.7%

Table 3: Activity transition detection and inference result for selected apps. All results are generated using Activity traces with more than 3000 transitions.

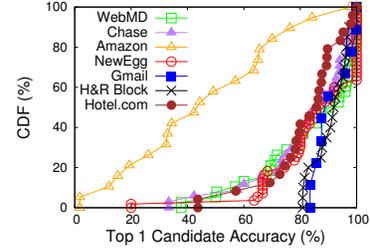


Figure 11: CDF of the average accuracy for top 1 candidates

5 Evaluation

In this section, we evaluate (1) the effectiveness of the automated Activity transition tool, (2) the performance of the Activity inference, and (3) the attack overhead.

Attack implementation. We implement the Activity inference attack with around 2300 lines of C++ code compiled with Android NDK packaged in an attack app.

Data collection. We use the automated tool in §4.3 to generate Activity transitions. We use random traversals to simulate user behavior, and deterministic traversals in controlled experiments for training and parameter selection, *e.g.*, the sampling rate. We run all experiments on Samsung Galaxy S3 devices with Android 4.2. We do not make use of any device-specific features and expect our findings to apply to other Android phones.

App selection and characteristics. In our experiments, we choose 7 Android apps, WebMD, GMail, Chase, H&R Block, Amazon, NewEgg, and Hotel.com, all of which are popular and also contain sensitive information. Table 2 characterizes these apps according to properties relevant to our attack techniques. NewEgg and GMail have the highest and the lowest number of Activities, and Amazon has the highest graph density. Chase app is the only one with no automatic soft keyboard pop-up during the transition among these apps. The Content Provider is only extensively used by WebMD. Except GMail, the percentage of the network feature is usually high.

5.1 Activity Transition Tool Evaluation

For Activity transition graph generation, the tool typically spends 10 minutes to 1 hour on a single Activity, depending on the UI complexity. For all apps except

WebMD, the generated transition graphs are exactly the same as the ones we generate manually. The transition graph of WebMD misses 4 *create* transition edges and 3 *resume* transition edges, which is caused by dependent Activity issues described in §4.3. Our tool generates no fake edges for all selected apps.

5.2 Activity Inference Attack Evaluation

Evaluation methodology. We run the attack app in the background while the tool triggers Activity transitions. The triggered Activity traces are recorded as the ground truth. To simulate the real attack environment, the attack is launched with popular apps such as GMail and Facebook running in the background. For the Activity transition detection, we measure the accuracy, false positive (FP) and false negative (FN) rates. For the Activity inference, we consider the accuracy for the top N candidates — the inference is considered correct if the right Activity is ranked top N on the result candidate list.

5.2.1 Activity Transition Detection Results

Aggregated Activity transition detection results are shown in columns 2–4 in Table 3. For all selected apps, the detection accuracies are more than 96.5%, and the FP and FN rates are both less than 4%.

When changing the sampling period from 30 to 100 ms in our experiment, for all apps the increases of FP and FN rates are no more than 5%. This shows a small impact of the sampling rate on the detection; thus, a lower sampling rate can be used to reduce sampling overhead.

We also measure Activity transition detection delay, which is the time from the first *shared_vm* increase to the moment when the Activity transition is detected in

Fig. 10. For all apps, 80% of the delay is shorter than 1300 ms, which is fast enough for Activity tracking.

5.2.2 Activity Inference Results

The aggregated Activity transition inference result is shown in column 5–7 in Table 3. For all apps except Amazon, the average accuracies for the top 1 candidates are 82.6–92.0%, while the top 2 and top 3 candidates’ accuracies exceed 91.4% and 93.6%. Amazon’s accuracy remains poor, and can achieve 80% only when considering the top 5 candidates. In the next section, we will investigate more into the reason of these results.

Fig. 11 shows the CDF of the accuracy for top 1 candidates per Activity in the selected apps. Except Amazon, all apps have more than 70% of Activities with more than 80% accuracy. For WebMD, NewEgg, Chase and Hotel.com, around 20% Activities have less than 70% accuracy. For these Activities, they usually lack some signature features, or the features they have are too common to be distinct enough. However, such Activities usually do not have sensitive data due to a lack of important UI features such as text fields for typing, and thus are not relevant to the proposed attacks. For example, in Hotel.com, the two Activities with less than 70% accuracy are CountrySelectActivity for switching language and OpinionLabEmbeddedBrowserActivity for rating the app.

5.2.3 Breakdown Analysis and Discussion

To better understand the performance results, we break down the contributions of each signature feature and the transition model further. Table 4 shows the decrease of the average accuracy for top 1 candidates if leaving out certain features or the transition model. Without the CPU utilization time feature, the accuracy decreases by 36.2% on average, making it the most important contributor. Contributions from the network feature and the transition model are also high, which generally improves the accuracy by 12–30%. As low-entropy features, the Content Provider and the input method contribute under 5%. Thus, the CPU utilization time, the network event and the transition model are the three most important contributors to the final accuracy. Note that though the Content Provider and input method features have lower contributions, we find that the top 2 and top 3 candidates’ accuracies benefit more from them. This is because they are more stable features, and greatly reduce the cases with extremely poor results due to the high variance in the CPU utilization time and the network features.

Thus, considering that the CPU utilization time is always available, apps with a high percentage of network features, or a sparse transition graph, or both, should have a high inference accuracy. In Table 2 and Table 3, this rule applies to all the selected apps except Amazon.

Application name	Δ Accuracy for top 1 candidates				
	no IM	no CP	no Net	no CPU	no HMM
WebMD	-3.8%	-2.6%	-19.1%	-25.7%	-16.6%
Chase	-0%	-2.0%	-12.8%	-71.5%	-28.7%
Amazon	-10.2%	-0%	-3.2%	-32.0%	-5.9%
NewEgg	-0.5%	-0%	-31.7%	-20.0%	-13.0%
GMail	-13.7%	-0%	-0.9%	-58.6%	-19.4%
H&RBlock	-0.7%	-0%	-30.7%	-27.9%	-16.5%
Hotel.com	-0.3%	-0%	-28.8%	-17.9%	-12.2%

Table 4: Breakdown of individual contributions to accuracy. IM, CP, Net, and CPU stand for input method, Content Provider, network event and CPU utilization time.

Amazon has a low accuracy mainly because it benefits little from either the transition model or the network event feature due to high transition graph density and infrequent network events. The reason for the high transition graph density is that in Amazon each Activity has a menu which provides options to transition to nearly all other Activities. The infrequent network events are due to its extensively usage of caching, presumably because much of its content is static and cacheable. However, we note that many network events are typically not cacheable, *e.g.*, authentication events and dynamic content (generated depending on the user input and/or the context). Compared to the other 6 apps, we find that these two properties for Amazon are not typical, not present in another shopping app NewEgg.

The Amazon app case indicates that our inference method may not work well if certain features are not sufficiently distinct, especially the major contributors such as the transition model and the network event feature. To better understand the general applicability of our inference technique, a more extensive measurement study about the Activity and Activity transition graph properties is needed, which we leave as future work.

5.2.4 Attack overhead

We use the Monsoon Power Monitor [28] to measure the attack energy overhead. Using an Activity trace of WebMD on the same device, with our attack in the background the power level increases by 2.2 to 6.0% when the sampling period changes from 100 to 30 ms.

6 Enabled Attack: Activity Hijacking

In this section, based on the UI state tracking, we design a new Android attack which breaches GUI integrity — Activity hijacking attack — based on a simple idea: stealthily inject into the foreground a phishing Activity at the right timing and steal sensitive information a user enters, *e.g.*, the password in login Activity.

Note that this is not the first attack popping up a phishing Activity to steal user input, but we argue that it is the first general one that can hijack any Activities during an app’s lifetime. Previous study [29] pops up a fake login

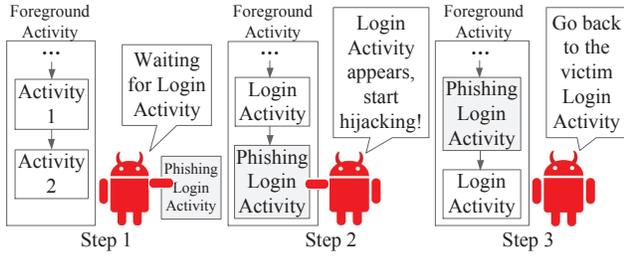


Figure 12: General steps of the Activity hijacking attack

Activity every time the attack app detects the launching of the target app, tricking users into entering login credentials. However, this can easily cause user suspicion due to the following: (1) most apps nowadays do not require login right after the app starts, even for banking apps like Chase; (2) the attack requires suspicious permissions such as `BOOT_COMPLETED` to be notified of system boot, based on the assumption that login is expected after the system reboot. With the Activity inference attack, we no longer suffer from these limitations.

6.1 Activity Hijacking Attack Overview

Fig. 12 shows the general steps of Activity hijacking attack. In step 1, the background attack app uses Activity inference to monitor the foreground Activity, waiting for the attack target, for example, `LoginActivity` in Fig. 12.

In step 2, once the Activity inference reports that the target victim Activity, *e.g.*, `LoginActivity`, is about to enter the foreground, the attack app simultaneously injects a pre-prepared phishing `LoginActivity` into the foreground. Note that the challenge here is that this introduces a race condition where the injected phishing Activity might enter the foreground too early or too late, causing visual disruption (*e.g.*, broken animation). With carefully designed timing, we prepare the injection at the perfect time without any human-observable glitches during the transition (see video demos [6]). Thus, the user will not notice any differences, and continue entering the password. At this point, the information is stolen and the attack succeeds.

In step 3, the attack app ends the attack as unsuspectingly as possible. Since the attack app now becomes the foreground app, it needs to somehow transition back to the original app without raising much suspicion.

6.2 Attack Design Details

Activity injection. To understand how it is possible to inject an Activity from one app into the foreground and preempt the existing one, we have to understand the design principle of smartphone UI. If we think about apps such as the alarm and reminder apps, they indeed require the ability to pop up a window and preempt any foreground Activities. In Android, such functionality is sup-

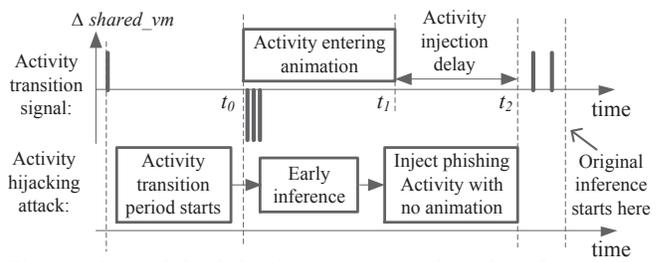


Figure 13: Activity injection process with early inference

ported in two ways without requiring any permissions: (1) starting an Activity with a restricted launching mode “`SingleInstance`” [30]; (2) starting an Activity from an Android broadcast receiver [31]. In our design, since the timing of the injection is critical, we choose the former as it can be launched 30 ms faster.

UI phishing. To ensure that the phishing Activity’s UI appears the same as the victim Activity, we disassemble the victim app’s apk using *apktool* [32] and copy all related UI resources to the attack app. However, sometimes the Activity UI may have dynamically loaded areas which are not determined by the UI resources, *e.g.*, the account verification image in banking apps. To solve that, the attacker can make those areas transparent, given that Android supports partially transparent Activity [33].

Activity transition animation modifying. Since our injection introduces an additional Activity transition which is racing with the original transition, the animation of the additional transition would clearly disrupt the flow. Fortunately, this problem can be solved by disabling the transition animation (allowed by Android) by modifying an Activity configuration of the attack app without needing any permissions. This helps the injection become totally seamless, and as will be discussed in §9, enforcing this animation may be a feasible mitigation of our attack.

Injection timing constraint. For the attack to succeed, the Activity injection needs to happen before any user interaction begins, otherwise the UI change triggered by it may be disrupted by the injected Activity. Since the injection with the current inference technique takes quite long (the injected Activity will show up after around 1300 ms from the first detected *shared_vm* increase as measured in §5), any user interaction during this period would cause disruptions. To reduce the delay, we adapt the inference to start much earlier. As shown in Fig. 13, we now start the inference as soon as the *shared_vm* decrease is observed (roughly corresponding to the Activity entering animation start time). In contrast, our original inference starts after the last *shared_vm* increase.

Note that this would limit the feature collection up to the point of the *shared_vm* decrease, thus impacting the inference accuracy. Fortunately, as indicated in Fig. 10, such change does allow the network event feature, the

majority of the CPU utilization time features, and the transition model to be included in the inference, which are the three most important contributors to the final accuracy as discussed in §5.2.3. Based on our evaluation, this would reduce the delay to only around 500 ms.

Unsuspecting attack ending. As described in §6.1, in step 3 we try to transition from the attack app back to the victim unsuspectingly. Since the phishing Activity now has the information entered on the screen, it is too abrupt to directly close it and jump back to the victim. In our design, we leverage “benign” abnormal events to hide the attack behavior, *e.g.*, the attack app can show “server error” after the user clicks on the login button, and then quickly destroy itself and fall back to the victim.

Deal with cached user data. It is common that some sensitive data may be cached, thus won’t be entered at all, *e.g.*, the user name in login Activity. Without waiting for them to expire, it is difficult to capture any fresh input.

Note that we can simply inject the phishing Activity with all fields left blank. The challenge is to not alert the user with any other observable suspicious behavior. Specifically, depending on the implementation, we find that the cached user data sometimes show up immediately in the very first frame of the Activity entering animation (t_0 in Fig. 13). Thus, our later injection would clear the cached fields, which causes visual disruption.

Our solution is to pop up a tailored cache expiration message (replicating the one from the app), and then clear such cached data, prompting users to re-enter them.

6.3 Attack Implementation and Evaluation

Implementation. We implement Activity hijacking attack against 4 Activities: H&R Block’s LoginActivity and RefundInfoActivity for stealing the login credentials and SSN, and NewEgg’s ShippingAddressAddActivity and PaymentOptionsModifyActivity for stealing the shipping/billing address and credit card information. The latter two Activities do not appear frequently in the check-out process since the corresponding information may be cached. Thus, to force the user to re-enter them, our attack injects these two Activities into the check-out process. The user would simply think that the cached information has expired. In this case the fake cache expiration messages are not needed, since the attack can fall back to the check-out process naturally after entering that information. Attack demos can be found in [6].

Evaluation. The most important metric for our attack is the Activity injection delay, which is the time from t_1 to t_2 in Fig. 13. In Android, it is hard to know precisely the animation ending time t_1 , so the delay is measured from t_0 to t_2 as an upper bound. In the evaluation the Activity injection is performed 50 times for the LoginActivity of H&R Block app, and the average injection delay is 488 ms. Most of the delay time is spent in `onCreate()`

(242 ms) and `performTraverse()` (153 ms). From our experience, the injection is fast enough to complete before any user interaction starts.

7 Enabled Attack: Camera Peeking

In this section, we show another new attack enabled by the Activity inference: camera peeking attack.

7.1 Camera Peeking Attack Overview

Due to privacy concerns, many apps store photo images shot by the camera only in memory and never make them publicly accessible, for example by writing them to external storage. This applies to many apps such as banking apps (*e.g.*, Chase), shopping apps (*e.g.*, Amazon and Best Buy), and search apps (*e.g.*, Google Goggles). Such photo images contain highly-sensitive information such as the user’s life events, shopping interests, home address and signature (on the check). Surprisingly, we show that with Activity tracking such sensitive and well-protected camera photo images can be successfully stolen by a background attack app. Different from PlaceRaider [34], our attack targets at the camera photo shot by the user, instead of random ones of the environment.

Our attack follows a simple idea: when an Activity is using the camera, the attack app quickly takes a separate image while the camera is still in the same position. In the following, we detail our design and implementation.

7.2 Attack Design Details

Background on Android camera resource management. With the camera permission, an Android app can obtain and release the camera by calling `open()` and `release()`. Once the camera is obtained, an app can then take pictures by calling `takePicture()`. There are two important properties: (1) exclusive usage. The camera can be used by only one app at any point in time; (2) slow initialization. Camera initialization needs to work with hardware, so `open()` typically takes 500–1000 ms (measured on Samsung Galaxy S3 devices).

Obtain camera images in the background. In the Android documentation, taking pictures in the background is explicitly disallowed due to privacy concerns [35]. Though PlaceRaider [34] succeeded in doing so, we find that their technique is restricted to certain devices running old Android systems which do not follow the documentation correctly, *e.g.*, Droid 3 with Android 2.3.

Interestingly, we find *camera preview frames* to be the perfect alternative interface for obtaining camera images without explicitly calling `takePicture()`. When using the camera, the preview on the screen shows a live video stream from the camera. Using `PreviewCallback()`, the video stream frames are returned one by one, which are actually the camera images we want. `SurfaceTexture` is used to capture this

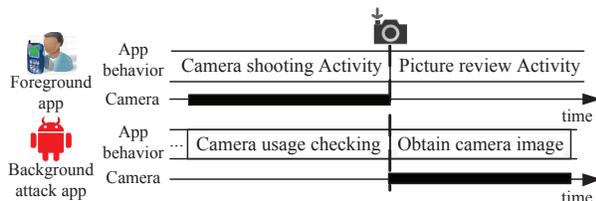


Figure 14: Camera peeking attack process when the foreground Activity is using the camera

image stream, and we find that it can be created with a nonexistent OpenGL texture object name, thus preventing any visible preview on the screen. We suspect that the less restrictive interface is managed by OpenGL library which bypasses the Android framework and its associated security checks. Compared to PlaceRaider [34], this technique not only has no requirement of the sensitive `MODIFY_AUDIO_SETTINGS` permission to avoid shutter sound, but also has much faster “shutter speed” of 24 frames per second. Note that even if this interface is blocked, our attack can still use techniques in §6 to inject an Activity to the foreground to get the preview frames.

Obtain photo images shot by the user. Fig. 14 shows how our attack gets the photo image the user shoots in the victim app. The photo taking functionality usually involves a camera shooting Activity and a picture review Activity. Once the user clicks on the shutter button in the former, the latter pops up with the picture just taken. Due to the exclusive usage property, when the foreground Activity is using the camera the attack app cannot get the camera. Thus, once knowing that the camera is in use, the attack app keeps calling `open()` to request the camera until it succeeds right after the user presses the shutter button and the camera gets released during the Activity transition. Since the delay to get a camera preview frame is only the initialization time (500–1000 ms), the camera is very likely still pointing at the same object, thus obtaining a similar image.

Capture the camera usage time. To trigger the attack process in Fig. 14, the attack app needs to know when the camera is in use in the foreground. However, due to the slow initialization, a naive solution which periodically calls `open()` to check the camera usage will possess the camera for 500–1000 ms for each checking action when the camera is not in use. During that time, if the foreground app tries to use the camera, a denial of service (DoS) will take place. With 12 popular apps, we find that when failing to get the camera, most apps pop up a “camera error” or “camera is used by another app” message and some even crash immediately. These errors may indicate that an attack is ongoing and alert the user. Besides, the frequent camera resource possessing behavior is easily found suspicious with increasing concerns about smartphone camera usage [34].

To solve the problem, our attack uses Activity infer-

Camera peeking attack type	Success rate	DoS rate	# of camera possession per round
Blind attack (3s idle time)	81%	19%	30.5
Blind attack (4s idle time)	83%	14%	20.9
Blind attack (5s idle time)	79%	8%	18.9
UI state based attack	99%	0%	1.4

Table 5: User study evaluation result for the camera peeking attack

ence to capture the camera usage time by directly waiting for the camera shooting Activity. To increase the inference accuracy for Activities using the camera, we add camera usage as a binary feature (true or false on the camera usage status) and it is only tested when the landing Activity is very likely to be the camera shooting Activity based on other features to prevent DoS and overly frequent camera possessions.

7.3 Attack Evaluation

Implementation. We implement the camera peeking attack against the check deposit functionality in Chase app, which allows users to deposit personal checks by taking a picture of the check. Besides the network permission, the attack app also needs the camera permission to access camera preview frames. On the check photo, the attacker can steal much highly-sensitive personal information, including the user name, home address, bank routing/account number, and even the user’s signature. A video demo is available at [6].

Evaluation methodology. We compare our UI state based camera peeking attack against the blind attack, which periodically calls `open()` to check the foreground camera usage as described in §7.2. We add parameter *idle time* for the blind attack as the camera usage checking period. The longer the idle time is, the lower the DoS possibility and the camera possession frequency are. However, the idle time cannot be so long that the attack misses the camera shooting events. Thus, the blind attack faces a trade-off between the DoS risk, the camera possession frequency, and the attack success rate.

User study. We evaluate our attack with a user study of 10 volunteers. In the study we use 4 Samsung Galaxy S3 phones with Android 4.1. Three of them use the blind attacks with idle time being 3, 4 and 5 seconds respectively, and the last one uses the UI state based attack. Each user performs 10 rounds, and in each round, the users are asked to first interact with the app as usual, and then go to the check deposit Activity and take a picture of a check provided by us. We emphasize that they are expected to perform as if it is their own check. The IRB for this study has been approved and we took steps to ensure that no sensitive user data were exposed, *e.g.*, by using a fake bank account and personal checks.

Performance metrics. For evaluation we measure: (1)

DoS rate, the ratio that when the user wants to use the camera but fails; (2) number of camera possessions, the number of events that the camera is possessed by the attack app; (3) success rate, the ratio that the attacker gets the check image after the user shoots the check.

Result. Table 5 shows the user study evaluation results. With the camera usage feature, the UI state based attack can achieve 99% success rate, and the only failure case is due to a failure in detecting the Activity transition. For the blind attack, the success rate is less than 83%, and when the idle time increases, the success rate increases then decreases. The increase is due to lower DoS probability, and the decrease is because the users usually finish shooting in around 4 seconds (found in our user study), so when the idle time increases to 5 seconds, the blind attack misses some camera shooting events.

UI state based attack causes no DoS during the user study. For the blind attack, the DoS rate is around 8–19%, and decreases when the idle time increases. Considering that a single DoS case may likely cause “sudden death” for the attack app, this risk is high, especially compared to the UI state based attack.

The camera possession number for the UI state based attack is also a magnitude lower. Every round, except the necessary one for camera shooting, the UI state based attack only needs 0.4 excessive camera possessions, which is mainly caused by inaccurate inference. For the blind attack, to ensure a high success rate, the camera possession number is proportional to time, making it hard to avoid suspicious frequent camera possessions.

Fig. 15 includes an average quality check image “stolen” from a real user, showing that the image is clear enough to read all the private information.

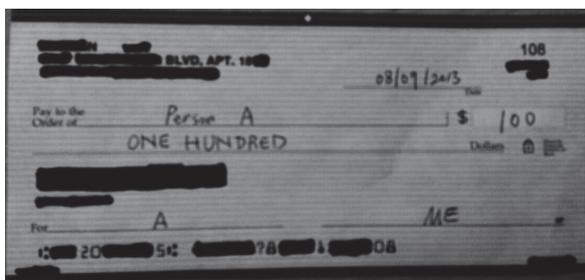


Figure 15: An example check image “stolen” using the camera peeking attack.

8 Other Attack Scenarios

Enhance existing attacks. Generally, a class of existing attacks that are launched only at specific timings benefits from UI state information. Since many attacks need to be launched at a specific timing, with the UI state information, both stealthiness and effectiveness can be achieved. For example, for the phishing attack using

TCP connection hijacking [9, 10], the attack app can precisely target at connections established in Activities with web pages instead of unrelated ones, *e.g.*, database updating, keepalive messages, *etc.* The attack thus becomes more efficient and less suspicious by avoiding frequently sending large amounts of attack traffic [9]. Similar enhancement can also be applied to keystroke inference attacks [7, 8] and screenshot taking attack [5] where only keystrokes entered in login Activities may be of interest.

User behavior monitoring and analysis. UI states themselves are user behavior related internal states of an app. As shown in Fig. 2, due to the limited screen size on the smartphone, full-screen window-level UI state information breaks user-app interaction to very fine-grained states. Thus, by tracking UI states, a background app can monitor and analyze user behavior, *e.g.*, in a health app the user is more often looking for drugs or physicians.

In addition, with Activity tracking, the attacker can even infer which choice is made inside an Activity (*e.g.*, which medical condition a user is interested in). This is achieved using the size of the request packet obtained by the technique described in §4.1. For example, for QAListActivity of H&R Block app, we can infer which tax question a user is interested in based on the length of the question that is embedded in the query packet. In this question list, we find that 10 out of 11 question queries are distinguishable (with different lengths).

A similar technique was proposed recently [12], but built upon a network event based state machine, with two limitations: (1) packet size itself can be highly variable (ads connections may co-occur) and different Activities may generate similar packet size traces, *e.g.*, login Activities and user account Activities both have the authentication connection thus may have similar packet size trace. UI state knowledge would limit the space of possible connections significantly as we infer the Activity based on a more comprehensive set of features; (2) not all user choices in Android are reflected in network packets — database/Content Provider can also be used to fetch content. With our UI state machine, we can further extend the attack to the Content Provider based user choice inference. For example, in WebMD, DrugSearchMainActivity has a list of letter A to Z. Once one letter is clicked, Content Provider is queried to fetch a list of drug names starting from that letter. With the Content Provider query event and content size inference technique (described in §4.1), we characterized all of the choices and found fairly good entropy: the responding content sizes have 16 different values for the 26 letters, corresponding to 4 bits out of 4.7 bits of information for the user choice.

9 Defense Discussion

In this section, we suggest more secure system designs to defend against the attacks found in this paper.

Proc file system access control. In our attack, *shared_vm* and features of Activity signature such as CPU and network all rely on freely accessible files in proc file system. However, as pointed out by Zhou *et al.* [12], simply removing them from the list of public resources may not be a good option due to the large amount of existing apps depending on them. To better solve the problem, Zhou *et al.* [12] proposed a mitigation strategy which reduces the attack effectiveness by rounding up or down the actual value. This can work for the network side channel, but may not be effective for *shared_vm* and *shared_pm*, which are already rounded to pages (4KB) but still generate a high transition detection accuracy. This is mainly because the window buffer size is large enough to be distinct and the side channel is pretty clean, as discussed in §3.3. Thus, Android system may need to reconsider its access control strategy for these public accessible files to better balance functionality and security. In fact, Android has already restricted access to certain proc files that are publicly accessible in the standard Linux, *e.g.*, */proc/pid/smaps*. However, our work indicates that it is still far from being secure.

Window manager design. As described in §3.2, the existence of the shared-memory side channel is due to the requirement of the window buffer sharing in the client-drawn buffer design. Thus, a fundamental way of defending against the UI state inference attack in this paper is to use the server-drawn buffer design in GUI systems, though this means that any applications that are exposed to the details of the client-drawn buffer design need to be updated, which may introduce other side effects.

Window buffer reuse. The Activity transition signal consists of *shared_vm* increases and decreases, corresponding to window buffer allocations and deallocations. To eliminate such signal, the system can avoid them by pre-allocating two copies of the buffers and reuse them for all transitions in an app. Note that this is at the cost of much more memory usage for each app, as each buffer is several megabytes in size. However, with increasingly larger memory size in future mobile devices [36], we think this overhead may be acceptable.

In this paper, the most serious security breaches are caused by follow-up attacks based on UI state inference. Thus, we provide suggestions as follows that can mitigate the attacks even if the UI state information is leaked.

Enforce UI state transition animation. Animation is an important indicator for informing users about app state changes. In the Activity hijacking attack in §6, the seamless Activity injection is possible because this indicator can be turned off in Android. With UI state tracking, the attacker can leverage this to replace the foreground UI state with a phishing one without any visible indications. Thus, one defense on GUI system design side is to always keep this indicator on by enforcing animation

in all UI state transitions. This helps reduce the attack stealthiness though it cannot fully eliminate the attack.

Limit background application functionality. In GUI systems, background applications do not directly interact with users, so they should not perform privacy-sensitive actions freely. In §7, a background attacker can still get camera images, indicating that Android did not sufficiently restrict the background app functionality. With UI state tracking, an attacker can leverage precise timing to circumvent app data isolation. Thus, more restrictions should be imposed to limit background applications' access to sensitive resources like camera, GPS, sensor, *etc.*

To summarize, we propose solutions that eliminate dependencies of the attack such as the proc file side channel, which may prevent the attack. However, more investigation is required to understand their effectiveness and most of them do require significant changes that have impact on either backward-compatibility or functionality.

10 Related Work

Android malware. The Android OS, like any systems, contains security vulnerabilities and is vulnerable to malware [37–39]. For instance, the IPC mechanisms leave Android vulnerable to confused deputy attacks [38, 39]. Malware can collect privacy-sensitive information by requesting certain permissions [37, 40]. To combat these flaws, a number of defenses have been proposed [38, 41], such as tracking the origin of inter-process calls to prevent unauthorized apps from indirectly accessing privileged information. Our attack requires neither specific vulnerabilities nor privacy-sensitive permissions, so known defense mechanisms will not protect against it.

Side-channel attacks. Much work has been done on studying side channels. **Proc file systems** have been long abused for side-channel attacks. Zhang *et al.* [24] found that the ESP/EIP value can be used to infer keystrokes. Qian *et al.* [10] used “sequence-number-dependent” packet counter side channels to infer TCP sequence number. In memento [11], the memory footprints were found to correlate with the web pages a user visits. Zhou *et al.* [12] found three Android/Linux public resources to leak private information. These attacks are mostly app-dependent, while in this paper the UI state inference applies generally to all Android apps, leading to not only a larger attack coverage but also many more serious attacks. **Timing** is another popular form of side channels. Studies have shown that timing can be used to infer keystrokes as well as user information revealed by web applications [23, 42–44]. **Sensors** are more recent, popular side-channel sources. The sound made by the keyboard [45], electromagnetic waves [46], and special software [47] can be used to infer keystrokes. More recently, a large number of sensor-based side channels have been discovered on Android, including the micro-

phone [18], accelerometer [7, 8] and camera [34]. Our attack does not rely on sensors which may require suspicious permissions. Instead, we leverage only data from the proc file system, which is readily available with no permission requirement.

Root causes of side-channel attacks. All side-channel attacks exist because of certain behavior in the software/hardware stack that can be distinguished through some forms of observable channels by attackers. For example, the inter-keystroke timing attack exploits the application and OS behavior that handles user input. SSH programs will send whatever keys the user types immediately to the network, so the timing is observable through a network packet trace [23]. For VIM-like programs, certain program routines are triggered whenever a new key is captured, so the timing can be captured through snapshots of the program’s ESP/EIP values [24]. The TCP sequence number inference attack [10] exploits the TCP stack of the OS that exposes internal states through observable packet counters. In our attack, we exploit a new side channel caused by popular GUI framework behavior, in particular how user interaction and window events are designed and implemented.

11 Conclusion

In this paper, we formulate the UI state inference attack designed at exposing the running UI state of an application. This attack is enabled by a newly-discovered shared-memory side channel, which exists in nearly all popular GUI systems. We design and implement the Android version of this attack, and show that it has a high inference accuracy by evaluating it on popular apps. We then show that UI state tracking can be used as a powerful attack building block to enable new Android attacks, including Activity hijacking and camera peeking. We also discuss ways of eliminating the side channel, and suggest more secure system designs.

Acknowledgments

We would like to thank Sanae Rosen, Denis Foo Kune, Zhengqin Luo, Yu Stephanie Sun, Earlece Fernandes, Mark S. Gordon, the anonymous reviewers, and our shepherd, Jaeyeon Jung, for providing valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1318306, CNS-1059372, CNS-1039657, CNS-1345226, and CNS-0964545, as well as by the ONR grant N00014-14-1-0440.

References

[1] N. Feske and C. Helmuth, “A Nitpickers guide to a minimal-complexity secure GUI,” in *ACSAC*, 2005.

- [2] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, “Design of the EROS trusted window system,” in *USENIX Security Symposium*, 2004.
- [3] T. Fischer, A.-R. Sadeghi, and M. Winandy, “A pattern for secure graphical user interface systems,” in *20th International Workshop on Database and Expert Systems Application*. IEEE, 2009.
- [4] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang, “A Systematic Approach to Uncover Security Flaws in GUI Logic,” in *IEEE Symposium on Security and Privacy*, 2007.
- [5] C.-C. Lin, H. Li, X. Zhou, and X. Wang, “Screenmilk: How to Milk Your Android Screen for Secrets,” in *NDSS*, 2014.
- [6] “Video Demos for this Paper,” <https://sites.google.com/site/uistateinferenceattack/demos>.
- [7] Z. Xu, K. Bai, and S. Zhu, “Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors,” in *WiSec*, 2012.
- [8] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, “Tappints: your finger taps have fingerprints,” in *Mobisys*, 2012.
- [9] Z. Qian and Z. M. Mao, “Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security,” in *IEEE Symposium on Security and Privacy*, 2012.
- [10] Z. Qian, Z. M. Mao, and Y. Xie, “Collaborative tcp sequence number inference attack: how to crack sequence number under a second,” in *CCS*, 2012.
- [11] S. Jana and V. Shmatikov, “Memento: Learning Secrets from Process Footprints,” in *IEEE Symposium on Security and Privacy*, 2012.
- [12] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, “Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources,” in *CCS*, 2013.
- [13] “Wayland,” <http://wayland.freedesktop.org/>.
- [14] “Ubuntu Move to Wayland,” <http://www.markshuttleworth.com/archives/551>.
- [15] “Mir,” <https://wiki.ubuntu.com/Mir>.
- [16] “Back Stack,” <http://developer.android.com/guide/components/tasks-and-back-stack.html>.

- [17] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *OOPSLA*, 2013.
- [18] 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*, 2011.
- [19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *CCS*, 2011.
- [20] "VMMMap," <http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx>.
- [21] "project-butter," <http://www.androidpolice.com/2012/07/12/getting-to-know-android-4-1-part-3-project-butter-how-it-works-and-what-it-added/>.
- [22] "Android DDMS," <http://developer.android.com/tools/debugging/ddms.html>.
- [23] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.
- [24] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.
- [25] "Whois IP Address Database," <http://whois.net/>.
- [26] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.
- [27] "Android Activity Testing," http://developer.android.com/tools/testing/activity_testing.html.
- [28] "Monsoon Power Monitor," <http://www.msoon.com/LabEquipment/PowerMonitor/>.
- [29] "Focus Stealing Vulnerability," <http://blog.spiderlabs.com/2011/08/twsl2011-008-focus-stealing-vulnerability-in-android.html>.
- [30] "Android Launching Mode," <http://developer.android.com/guide/topics/manifest/activity-element.html#lmode>.
- [31] "Android Broadcast Receiver," <http://developer.android.com/reference/android/content/BroadcastReceiver.html>.
- [32] "Android Apktool," <http://code.google.com/p/android-apktool/>.
- [33] "Transparent Activity Theme," <http://developer.android.com/guide/topics/ui/themes.html#ApplyATheme>.
- [34] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, "PlaceRaider: Virtual Theft in Physical Spaces with Smartphones," in *NDSS*, 2013.
- [35] "Android Camera," <http://developer.android.com/reference/android/hardware/Camera.html>.
- [36] "Samsung Wants to Cram 4GB of RAM into Your Next Phone," <http://www.pcworld.com/article/2083320/samsung-lays-groundwork-for-smartphones-with-more-ram.html>.
- [37] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, 2012.
- [38] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-delegation: Attacks and Defenses," in *USENIX Security Symposium*, 2011.
- [39] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," in *NDSS*, 2013.
- [40] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information Flow Tracking System for Real-Time Privacy Monitoring on Smartphones," in *OSDI*, 2010.
- [41] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *USENIX Security Symposium*, 2011.
- [42] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *IEEE Symposium on Security and Privacy*, 2010.
- [43] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, 2012.
- [44] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *WWW*, 2007.
- [45] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," in *CCS*, 2005.
- [46] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," in *USENIX security symposium*, 2009.
- [47] K. Killourhy and R. Maxion, "Comparing Anomaly-Detection Algorithms for Keystroke Dynamic," in *DSN*, 2009.