PERUIM: Understanding Mobile Application Privacy with Permission-UI Mapping

Yuanchun Li, Yao Guo*, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
{yli, yaoguo, cherry}@pku.edu.cn

ABSTRACT
Current mobile operating systems such as Android employ the permission-based access control mechanism, but it is difficult for users to understand how and why the permissions are used within a particular application. This paper introduces *permission-UI mapping* as an easy-to-understand representation to illustrate how permissions are used by different UI components within a given application. Connecting UI components to permissions helps users to understand the purpose of permission requests and also makes it possible to illustrate permission requests in a fine-grained manner. We propose PERUIM to extract the permission-UI mapping from an application based on both dynamic and static analysis, and represent the analysis results with a graphical representation. Experiments on popular mobile applications demonstrate the accuracy and applicability of the proposed approach.

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation (e.g. HCI): User Interfaces; D.4.6 Operating Systems: Security and Protection

Author Keywords
Mobile applications; Android; functionality; permission; user interface (UI)

INTRODUCTION
In recent years, mobile applications (*apps* in short) have seen widespread adoption, with over one million apps available for download in both Google Play and Apple App Store, while billions of downloads have been accumulated [33, 34]. Many apps regularly request to access private user information such as contact lists, locations and photos. Some of these requests are warranted and necessary, while others may be exploited for malicious or advertising purposes.

As the most popular mobile operating system, Android employs a permission-based mechanism to control whether an app is allowed to access certain sensitive resources. When a user installs an app, he or she has the opportunity to review the list of permissions requested by the app and either agree to install, or cancel the installation if the permissions are excessive or objectionable [10].

Beginning in Android 6.0, users are able to grant permissions to an app at runtime after the app has been installed [6]. It also gives the user more fine-grained control over an app's ability to access sensitive data. For example, a user could choose to grant a camera app the permission to access camera but not the permission to access location. The user can also revoke each permission at any time in a later time.

Many accesses to sensitive data are undesirable from the end users' perspective. Previous work has shown that users are willing to block accesses to protected resources a third of the time under realistic circumstances [32]. However, it is difficult for most end users to understand how the Android permission system works; controlling the permissions for each app will be even more difficult [10].

On one hand, users often feel difficult to understand why and how each permission is required by the apps. Many researchers have proposed ideas about connecting permissions to other features understandable by users. WHYPER [19] and AutoCog [21] mapped permissions to textual descriptions, however the descriptions provided by developers are not security-centric and are sometimes significantly deviated from the permissions [37]. Lin *et al.* [14] and Wang *et al.* [30] connected permissions to third-party libraries or code segments in order to understand the purpose of permission requests. However, these approaches are mostly more developer-oriented than end-user-oriented.

On the other hand, each permission requested by an app might be used for multiple purposes within the same app. For example, location data can be used for both navigating and advertising. It would be desirable if a user can choose to grant location permission to the navigating purpose in an app, while disallowing it from using location for advertising purposes. Currently, not only there are no mechanisms to control these accesses in such a fine-grained manner, there are even no intuitive ways to represent such kind of situation and illustrate it to users in a straightforward representation.

In order to help end users better understand the permissions requested in an app, especially on why and how these permissions are used, this paper introduces *permission-UI mapping*, which connects permissions with user interface (UI) compo-

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UbiComp '16, September 12–16, 2016, Heidelberg, Germany
©2016 ACM. ISBN 978-1-4503-4461-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2971648.2971693

Figure 1. Two screenshots from the *com.devexpert.weather* app.

nents. Our goal is to present users how each permission is used within an app based on the granularity of UI components, such that a user can easily tell which UI components are using certain permissions.

Compared to previous approaches, a permission-UI mapping offers several benefits to help users understand permission requests. First, UI is more intuitive than libraries or code segments (e.g., classes). Users might feel easier to relate or infer the functionality of a UI component based on experiences or common sense. For example, if we show location is used in an advertising UI, the user can easily recognize the advertisement and realize whether it should be allowed. Second, UI provides a finer granularity such that users can distinguish between the permission requests for different purposes within the same app. Finally, it also gives us the potential to control permission accesses based on UI components, however this is out of the scope of this paper.

For example, Figure 1 shows two screenshots of the "Weather & Clock Widget Android" app (whose package name is *com.devexpert.weather*.). The app requests the LOCATION permission, which is perfectly fine as the app offers weather and map services, both of which are location-based. However, other components within the app also use the location information. For example, the advertisements displayed on the bottom of the screens are typically location-based too.

In order to make users aware of this situation, we can analyze the permission-UI mapping and present the permissions used in an app with UI-based visualization, as depicted in Figure 2. The figure shows how each permission is used by each UI component. We can also distinguish between which permissions have been accessed in order to render the UI component, and which permissions will be accessed if you click the UI component (if it is clickable). We choose to group the mappings by permissions instead of UI components, because there are only a few sensitive permissions concerned by users.

With the illustration in the figure, users can easily understand that the LOCATION permission is used by different components within the app, and may also take further actions to control whether the permission should be accessed by each UI components.

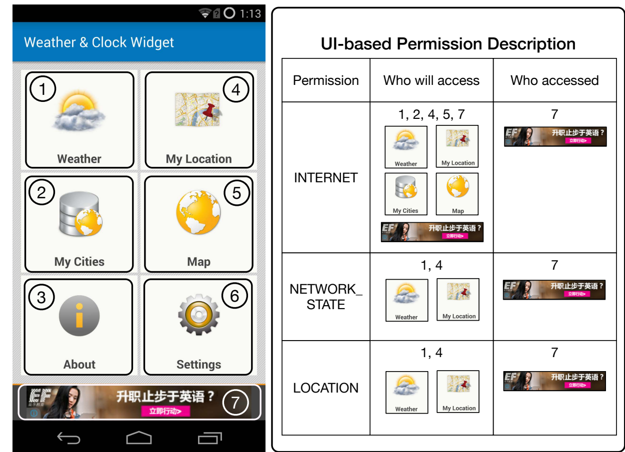

Figure 2. An example of permission-UI mapping. ***Note that this is a mock-up of the actual interface, the boxes and numbers are manually added for illustration only.***

In order to meet this goal, this paper proposes PERUIM, a mechanism to generate and visualize permission-UI mappings automatically. The key techniques proposed in PERUIM include: extracting UI components with an app through dynamic testing of the app on a customized Android image to form a UI transition graph, extracting permissions required by each UI component using both dynamic and static analysis, and inferring permission-UI mappings based on the UI transition graph.

To evaluate the applicability and scalability of PERUIM, we tested PERUIM on 200 popular Android apps from Google Play. PERUIM successfully analyzed 164 of them, while most failures were due the inherent limitations of automated dynamic analysis to handle special inputs such as logins.

We manually evaluated the accuracy of PERUIM on 10 popular apps from Google Play. PERUIM was able to extract 215 permission-UI mapping relations from the 10 apps. We then manually labeled the correct ones to examine its accuracy. The result shows that PERUIM achieves a 76.75% overall precision in inferring which permissions are used by each UI component. Although there is still space for improvement, the accuracy of PERUIM is better than most existing permission explanation approaches based on NLP.

This paper makes the following research contributions:

- We introduce the idea of permission-UI mapping to explain the permissions in Android apps. Compared to textual descriptions and code-level concepts, UI-based descriptions are arguably easier to understand for end users.
- We propose PERUIM, a tool taking advantage of both dynamic and static analysis to generate and visualize permission-UI mappings automatically, which can be directly applied on most popular apps.
- We evaluated PERUIM on popular Android apps and demonstrated that the permission-UI mappings produced by PERUIM can describe fine-grained permission requests in Android apps with an acceptable accuracy.

RELATED WORK

Permission Description

Many researchers have proposed approaches to help users or developers understand permissions used in Android apps. They can be categorized into text-description based, library-based, code-based and API-based approaches.

WHYPER [19] and AutoCog [21] apply NLP (natural language processing) techniques to analyze textual descriptions of apps and connect an app's descriptions to its requested permissions. The textual descriptions are easy for users to understand, but most descriptions cannot be fully mapped to permissions. One of the main reasons is that the descriptions are often provided by developers who do not have the incentive to explain the permission requests in detail.

Some approaches proposed the idea of understanding the purpose of permission requests based on code-level entities such as third-party libraries used in apps. Lin *et al.* [14, 15] introduced the idea of inferring the purpose of a permission by analyzing what third-party libraries an app uses. They categorized the purposes of several hundred third-party libraries (advertising, analytics, social network, etc.) and used crowdsourcing to ascertain people's level of concern for data use (e.g. location for advertising versus location for social networking). Wang *et al.* [30] extracted the textual information from custom code, such as method names, variable names and annotations, then applied supervised machine learning to predict the purposes of these code.

Zhang *et al.* [37] proposed the idea of automatically generating security-centric descriptions based on program analysis. They extracted *security behavior graphs* as high-level program semantics and applied *natural language generation* techniques to generate permission-related textual descriptions. However, the generated descriptions are about the data flow behavior in the app code, which may be too abstract for normal users.

There are also approaches aimed at generating API-permission mapping relations, such as PScout [2], STOWAWAY [9] and COPES [3]. They are very helpful for developers and researchers. Our work also makes use of the API-permission mapping result of PScout to help extracting permissions through static analysis. However these approaches are not intended for normal users because APIs are not designed for them.

Permission-UI Relation Analysis

Many researchers have explored or considered the relation between permissions and user interface in Android apps. Most of them are interested in the gap between UI and permissions, which might indicate stealthy behaviors, while others attempt to identify sensitive input views that can be considered as sources in taint analysis.

AsDroid [12] identifies the permissions related to UI components using static analysis and detects malware by detecting mismatches between the permissions and the text extracted from UI components. AppIntent [36] regards the data transmissions without user intention as likely privacy leakage. For each data transmission, AppIntent is able to provide a sequence of GUI manipulations corresponding to the events that lead to the transmission. Rubin *et al.* [25] also used static analysis to detect covert network accesses, in which they consider the network accesses without user awareness as covert.

Another type of related work focused on the detection of sensitive UI components, such as SUPOR [11] and UIPicker [18]. They detect sensitive input UI views in apps through machine learning and identify their related code in order to facilitate taint analysis. They regard user inputs as sensitive information and connect the input windows to the corresponding code entries, in order to detect leakage of user input through taint analysis. Their main goal is to estimate the sensitivity of UI components, instead of mapping UI components to permissions.

Permission Isolation

The ultimate goal for users after understanding permissions is to control the permission accesses, which requires isolating permissions for each UI components.

The first type of permission isolation is library isolation. Most of these approaches focus on isolating libraries, especially ad libraries, to avoid permissions granted to trusted apps used by untrusted libraries. Approaches such as AdSplit [27], AFrame [38], NativeGuard [29] and AdDroid [20] put ad libraries in a separated process, while NativeGuard [29] focuses on native libraries. Compac [31] and FlexDroid [26] provided in-app privilege separation where libraries and the host app run in the same process.

The concept of UI-based isolation has been proposed by Roesner *et al.* [24]. They introduced user-driven access control as a mechanism of permission granting in modern operating system. They also proposed User Interface Toolkit Mechanisms [22], a programming toolkit to help developers design apps whose permissions are isolated at the UI component level. LayerCake [23] isolates user interface (UI) libraries from its host app to support secure third-party UI embedding on Android.

To help users make permission control at the UI component level is attractive if they can understand the permissions granted to each UI component. Our work on permission-UI mapping illustrates the permissions used by each UI component, which can be used as the foundation of UI-based permission control.

OUR APPROACH

This paper proposes PERUIM, an automated approach to generate permission-UI mappings in order to help users understand the permission requests within an Android app. Figure 3 shows the overall process of our approach. Given an Android app, PERUIM works in the following steps:

1. *Dynamic data extraction*. We first extract the UI data (including UI states and UI events), runtime permission accesses and runtime event handler callbacks. Dynamic analysis could be done manually to ensure higher coverage or automatically if applied to a large number of apps.
2. *UI modeling*. Using the extracted UI data, we build a UI state transition graph as a formal representation of the relationship among the UI states in a given app. It is based on a graph-based UI model, whose goal is to mend the gap between users' knowledge and abstract permission requests.

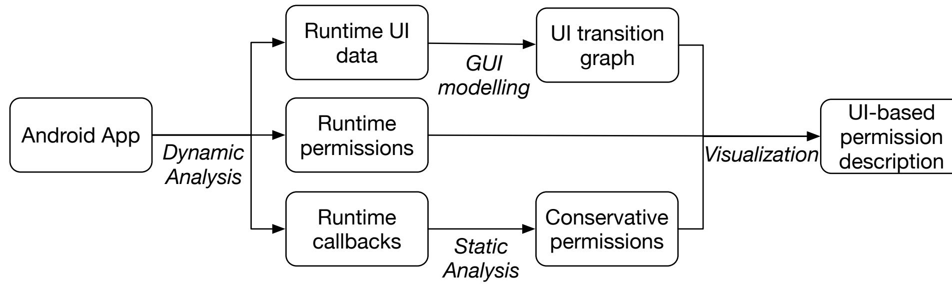

Figure 3. Overview of the PERUIM approach.

3. *Permission mapping*. In this step, we use both dynamic and static analysis to map the permissions to the UI model generated in Step 2. The UI events are where permissions meet the UI model. After this step, we have a UI state transition graph whose edges are tagged with permissions.

4. *Visualization*. Finally, we convert the permission-UI mapping model to a graphical representation similar to the mock-up shown in Figure 2. For each UI component, we inform users about which permissions are used to render its content and which permissions it will use.

GUI Modelling

User interface (UI) is the space where interactions between humans and machines occur [35]. In Android, an app's UI is everything that the user can see and interact with [7]. App developers design UI to help users understand the features of their apps, and users interact with the apps through the UI.

We assume that users are aware of the purpose of their interactions. Actually, unlike the permissions and code which can be confusing or easily obfuscated, developers typically would try their best to attract users by making the UI simple and easy-to-understand. Based on the assumption, we believe UI is one of the best choices to describe the often-confused permission requests.

UI States, Components and Events

Each UI page (or screenshot) presented in an Android app has a structured layout. All UI elements in an Android app are built using *View* and *ViewGroup* objects. A *View* is an object that draws something on the screen that the user can interact with. A *ViewGroup* is an object that holds other *View* objects in order to define the layout of the interface. The *View*s and *ViewGroup*s form a tree hierarchy, where *View*s are leaf nodes responsible for input controls and content rendering.

An app can be viewed as a combination of many states, each of which serves different functionality or renders different content. The tree structures of different states differ from each other, and the tree structures of the same state are similar. Thus we use the tree structure of a UI page to identify a UI state, and split the nodes in the tree structure into UI components.

DEFINITION 1. *A* ***UI state*** *represents a visual state of an Android app, which is shown to users in some situations, serving functionality or displaying content. A UI state can be represented with a tree data structure, where nodes are ViewGroups and Views.*

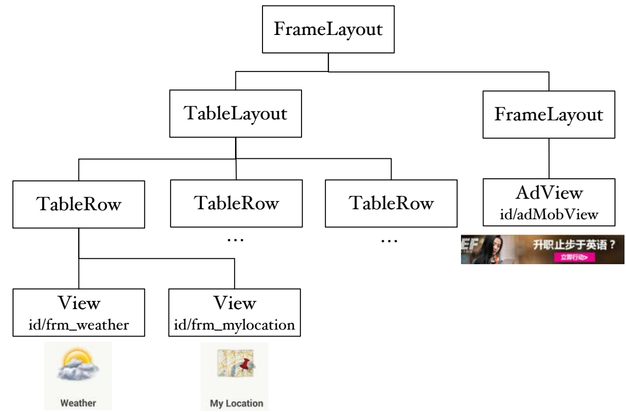

Figure 4. Simplified UI hierarchy of the screenshot(left) in Figure 1.

A UI state can be split into several UI components, each of which is responsible for serving one functionality or rendering one type of content.

For example, Figure 4 shows the simplified UI tree hierarchy of the screenshot (on the left) in Figure 1. The UI state represented by the tree hierarchy serves six functionalities and renders one content, for example the *id/frm_weather* node is a UI component representing the "Weather" functionality and the *id/adMobView* component renders advertisement.

Users are able to understand the meaning of each component in a UI state and decide how to interact with it. The way for users to interact with an app is using touchscreen gestures, such as touching, sliding and pressing buttons.

Android apps react to the gestures by registering event listeners to the corresponding UI components. After registering, the UI component will keep monitoring gestures and capturing the gestures if matched. Each event listener will have to implement gesture handling methods, which are callback methods invoked once a gesture is captured. We regard a UI event as a gesture which triggers a callback method. Normally a UI event will lead to transition from one UI state to another.

DEFINITION 2. *A* ***UI event*** *is a user's interaction that is handled by an Android app. It can be represented as a tuple* $E = < Gesture, Component, Handler >$ *where Gesture is the physical operation performed by a user, Component is the UI component that handles the gesture, and Handler is the callback method triggered by the UI event.*

For example, in the UI state shown in Figure 4, if the user want to learn about weather, he or she would click the "Weather"

UI component, by using a *Touch* gesture. In order to handle the gesture, the app should register an event listener to the "Weather" UI component. The *Touch* gesture, the touched "Weather" UI component and the `onClick` method together represent a UI event. The code skeleton about registering the event listener is shown below:

```
weather_btn=findViewById("id/frm_weather");
weather_btn.setOnClickListener(
 new OnClickListener {
   void onClick(View view) {
     //handle the event
     //start an Activity to display weather
   }
 }
)
```

UI Transition Graph

The user interface of an app can be viewed as a state machine. Each state is a page seen by users, and each edge is a UI that triggers the transition from one state to another. We define a structure named "UI transition graph" to represent the UI model for an app.

DEFINITION 3. *The user interface of an Android app is represented using a directed graph $G = (N, E)$ called* ***UI transition graph****. Each node $n \in N$ represents a UI state. The tree structures of the nodes in N are different from each other. Each edge $e \in E$ represents a UI event, which means the UI event leads to a state transition from the source node state to the target node state.*

For example, Figure 5 shows a partial UI transition graph of the motivating example app. There are four nodes in the partial UI transition graph, representing four UI states including the *Idle* UI state, *Menu* state, *Weather* state and the *Ad* state. The edges represent the UI events leading to transitions between UI states, for example the edge from state 2 to state 3 means that users can switch from the *Menu* state to the *Weather* page by touching (clicking) the "Weather" button.

So far we have modeled the GUI of an Android app with the *UI transition graph*, which serves as the bridge between *UI* and permissions. In the following steps we will connect permissions to the UI transition graph.

Permission Mapping

After constructing the UI transition graph of an Android app, PERUIM connects permissions to UI by mapping the permissions to the corresponding UI events.

Choosing UI events as the bridge between permissions and UI is based on the insight that the control flow of an app begins at event handlers. Since Android is event-driven, an app runs in a transient state when there is no event and starts complex logic when handling an event, thus most permission accesses would be triggered by UI events.

We use both dynamic and static analysis to generate precise and conservative mapping relations. Specifically, using dynamic analysis to extract "precise" results, i.e. permissions required at runtime, and using static analysis to generate "conservative" results, i.e. permissions possibly required under other conditions (in other branches that are not executed).

Figure 5. A partial UI transition graph for the *com.devexpert.weather* app.

Runtime Permission Mapping

We use dynamic analysis to extract runtime permission accesses. We instrumented the Android system to monitor permission accesses while running the app. Android checks permissions once there are sensitive APIs invoked, thus we can monitor the permission requests by logging at the permission checkpoints.

To connect the runtime permission requests to UI states and UI events, we also monitor the UI rendering behaviors and UI input events. The permissions accessed between a UI event and the rendering of its corresponding UI state are regarded as the permissions mapped to the UI event.

However, there are some background permission requests that may be mapped to UI events by mistake. For example, a navigation app running a background service to keep the location updated (requires the *Location* permission), and a communication app running a push service to frequently check arrived new messages (requires the *Internet* permission). We eliminate the incorrectly mapped permissions based on the observation that background permission accesses are often periodic. One reason is that some of the permission-related APIs are required to be used in a periodic manner, for example the location API `LocationManager.requestLocationUpdate` requires an argument `rate`, which is the frequency of location updates, and many background services are programmed in periodic patterns, for example:

```
while (True) {
  // check permission
  ...
  // sleep a few seconds
  sleep();
}
```

Thus most of the background permission requests are repeated periodically, which are easier to identify.

We identify these background permission requests by detecting periodic patterns. By tagging the runtime permission requests with timestamps, the problem of finding background permissions is converted to a problem of finding partial periodic patterns in time series in presence of noises and imprecise time information. Many researchers [17, 13] have attempted to address this problem. We directly applied one of the algorithms to detect and eliminate background permission requests.

The advantages of runtime permission extraction include:

- The runtime permissions are accurate because they are actually accessed, in comparison some of the static permissions are probably not reachable.
- Dynamic analysis is good at handling asynchronous calls, reflections and IPCs (inter-process communications) etc., which are difficult for static analysis.

However, the effectiveness of dynamic analysis relies on the coverage. It cannot extract permissions in the un-traversed code. For example, if the permission access code is in one branch and our dynamic analysis took another branch, the permissions will simply be missed by dynamic analysis, while in reality the permissions are related to the UI if the branch condition changes.

Static Permission Analysis

As a complement to dynamic analysis, we add static analysis to extract the permission requests missed during dynamic analysis.

Static program analysis is conservative because it analyzes all branches and considers all possible execution paths. Compared to dynamic analysis, the effectiveness of static analysis is not influenced by the branch conditions.

Android apps are event-driven. Unlike normal Java programs, Android apps have multiple entry points instead of a single `main` method, and each entry point is used to handle an event. The classes that handle the events are called *event listeners* and the methods handling the events are called *event handlers*. We mentioned earlier that user interactions are UI events sent to Android apps and the apps react to the events by registering event listeners.

Our static analysis is based on the runtime data extracted during dynamic analysis. We record the invocations of event handlers when running the app, thus we know the entry point methods corresponding to each user interaction. By constructing the call graph from the extracted entry point method, we are able to extract code related to the user interaction. The conservativeness is guaranteed in the call graph construction process.

After extracting the code related to a user interaction, we are able to extract the APIs invoked from the handler method. APIs are provided by the Android operating system and their required permissions are static. We use the permission-API mapping relations from PScout [2] and find the permission list for each UI event.

Note that the permissions found using static analysis are not completely conservative due to the weakness of static analysis. Some features in Java or Android are difficult for the static analyzer to deal with, such as asynchronous calls, IPC and reflection, which are frequently used in Android apps.

By combining the static analysis results with the dynamic analysis results, we get an accurate and conservative permission list for each UI event. So far, we have connected permissions to our UI model (UI transition graph), which can be used to generate UI-based permission description in the next step.

Visualization

The goal of our approach is to help users better understand the permissions in Android apps, thus we visualize our result and generate a UI-based permission description.

In this step, we use the UI transition graph as the representation of the UI of Android apps and map permissions to entry-point methods through program analysis. To mend the gap between UI and permissions, user events are used as the bridge between them. First, the user events are represented as the edges in the UI transition graph, because the user events lead to transitions between UI states. Then, the user events are mapped to entry-point methods, because the entry-point methods are what event listeners use to handle user input. Thus we are able to map permissions to the UI events and generate an artifact called *permission-tagged UI transition graph*, which is an extension of the UI transition graph, with the edges tagged with the permissions triggered by the UI events.

Unlike the Android system and in most app markets, where a list of requested permissions are shown to users before installing an app, PERUIM shows a list of permissions as well as the information on how the permissions are related to different UI components. For each permission, we tell users two things:

- Which UI components will access the permission if a user interacts with it, i.e. "who will access" the permission. For example, clicking a "Weather" button will use the *Location* permission.
- Which UI components are rendered after the permission is accessed, i.e. "who accessed" the permission. For example, an advertisement view accessed the *Location* permission.

For the "who will access" set, the permission-tagged UI transition graph can be directly used to calculate which permissions will be accessed after each user interaction. For example, an edge in the UI transition graph is tagged with the *Location* permission, and the gesture related to the UI event on the edge is a *Touch* gesture on the "Weather" view (just like the edge from state 2 to state 3 in Figure 5), thus the "Weather" view belongs to the "who will access" set of the *Location* permission.

The "who accessed" set cannot be directly inferred from the permission-tagged UI transition graph, thus we use data flow analysis to determine if sensitive data flows to the content of UI components.

Android provides a set of view content updating APIs, which are used for rendering dynamic information to users. The content is passed to these UI components as arguments of content updating APIs. Take the weather View as an example, the

Table 1. Common view content updating APIs in Android.

Class	Methods
android.view.View	onDraw setBackground setBackgroundResource
android.widget.TextView	setText append
android.widget.Toast	setText makeText
android.webkit.WebView	loadData loadDataWithBaseURL loadUrl
android.widget.ImageView	setImageBitmap setImageDrawable setImageIcon setImageResource setImageURI

app gets the user's location, sends it to server and receives a response containing the weather information, and invokes the `TextView.setText` method to display the weather information to the user.

We categorize the UI components in a UI state into two sets: static views that must have not used permissions (*MUST_NOT* set) and dynamic views which may have used permissions (*MAY* set). The *MUST_NOT* set can be calculated through a constant propagation analysis, i.e. checking if the arguments of the content updating APIs are constant. For example, the content of a TextView is set by the `TextView.setText` method, and we notice that the argument of the method is a constant string, thus we can conclude that the TextView is a static view that must have not used any permissions. The content updating APIs considered in PERUIM are shown in Table 1.

Note that we had another option in categorizing the views in one UI state, i.e. categorizing them to the *MUST* set that have used permissions and the *MAY_NOT* set that possibly have not used permissions. This could be done through taint analysis, i.e. marking sensitive data with taint tags, propagating the taint tags along program execution and checking whether the arguments of content updating APIs are tainted. We did not use this method because both static taint analysis [1] and dynamic taint analysis [8] lack scalability and accuracy, and also because we want to present a more conservative result to the users.

Finally, after calculating the *who will access* and the *who accessed* sets, we are able to generate visualized permission descriptions similar to the graph shown in Figure 2.

IMPLEMENTATION

UI Extraction

To extract the UI hierarchy, we make use of Android debugging tool ADB (short for Android Debug Bridge [4]) and Hierarchy Viewer [5].

Hierarchy Viewer is able to dump the layout of the current UI state on the screen, which is organized in a tree structure. Each UI component is a node in the tree hierarchy and is labeled with the resource ID, text and bounds. We can easily convert the Hierarchy Viewer result to our UI state nodes.

To extract the user interactions, we instrumented the Android system to monitor the UI events captured during runtime. We are interested in two things: which UI component handles the user interaction and which UI component is rendered after the user interaction.

When a user gesture is sent to a device, the device generates a TouchEvent with the coordinates on the screen. The TouchEvent is dispatched from the top down. At first, the TouchEvent is captured by the root node, then the root node dispatches the TouchEvent to its child nodes by calling `dispatchTouchEvent`. The TouchEvent stops being dispatched until it is handled by a node. We instrumented the `dispatchTouchEvent` method in the `View` class to capture the position of user gesture and find which UI component handles the gesture.

Similarly, the UI rendering process also follows a top-down manner. The parent nodes render themselves by calling the `draw` method and then trigger the rendering of their children. To monitor when and which UI components are rendered, we also instrumented the `View.draw` method.

After extracting the runtime data, we have collected a sequence of UI states and events. Finally we generate the UI transition graph by eliminating redundant UI states and connecting states with the corresponding events.

Permission Extraction

As stated earlier, we use both dynamic analysis and static analysis to extract the permission requests of Android apps.

We first extract runtime permission requests through dynamic analysis. In Android, the permission checking process is controlled in the `PackageManagerService` class. We instrument the `checkPermission` methods and the `checkUidPermission` methods in the `PackageManagerService` class, thus each permission request will be logged at runtime.

We use the open-source test input generation tool DroidBot [16] to perform dynamic analysis automatically. DroidBot remembers the UI states it reached and the UI components it touched and tries to explore as many UI states as possible. We extend DroidBot to record UI states and events, which are necessary for constructing the UI transition graph.

The effectiveness of PERUIM relies heavily on the coverage of dynamic analysis. The more states reached in dynamic analysis, the more detailed are the UI-based permission descriptions. However, improving test coverage is not our main goal, while the descriptions generated using DroidBot are helpful although it cannot achieve 100% coverage.

Then we generate a more conservative result using static analysis as a complement to dynamic analysis. As stated above, Android apps implement event listeners to handle user inputs. The event listeners are registered to the corresponding Views. When a View captures a user input, it checks if there are listeners registered, and invokes the handler method if there are

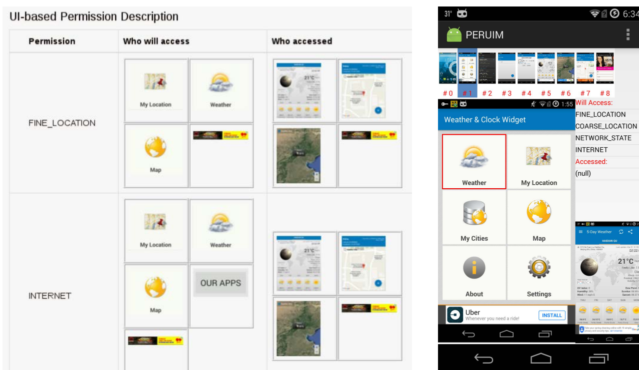

Figure 6. Example visualization interfaces of permission-UI mapping, including a web interface (left) and a mobile interface (right).

registered listeners. We modify the `View` class to record the handler methods registered to the View and invoked at runtime. These methods are entry-points of our static analysis.

We extend Soot [28] to construct the call graph from the entry-point methods and extract the APIs invoked in the methods. Finally we map the APIs to permissions using the mapping table from PScout [2]. To improve scalability, we skipped alias analysis because it is memory-hungry and time-consuming.

Permission-UI Mapping Visualization

In order to help users understand the permission-UI mapping, we implemented two prototypes to visualize the result of PERUIM: one using webpages and an Android app. Figure 6 shows the example visualization interfaces of PERUIM generated for the *com.devexpert.weather* app.

The web interface is designed to give users a big picture of the permission-UI mapping. It mainly contains a table in similar format with the mock-up interface in Figure 2. Each row in the table represents a permission, and each table cell contains the images of the UI components related to the permission. In order to help users understand the UI components, the webpage also presents a set of selected screenshots which contain the UI components (screenshots are omitted in Figure 6).

Due to the smaller screen, the mobile app interface provides an interactive way for the users to view the mappings grouped by UI components. As shown in Figure 6, users are able to select a screenshot and interact with it by touching a UI component, then the app will show the permissions related to the UI component and a screenshot of the UI state that will be triggered by the touch event.

EVALUATION

We then evaluate PERUIM with popular Google Play apps. We conduct experiments to answer the following research questions:

- How is the applicability and scalability of PERUIM?
- How many permissions could be explained by the PERUIM descriptions?
- Is the permission description generated by PERUIM accurate?

Table 2. Applicability of PERUIM. Note that the "Fail(D)" and "Fail(S)" columns mean the amounts of failures in dynamic analysis and static analysis respectively.

Category	App count	Fail(D)	Fail(S)	Success
Weather	50	4	0	46
Transportation	50	14	1	35
Tools	50	8	2	40
News	50	7	0	43
Total	200	33	3	164

Experiment Setup

We implemented PERUIM using Java and Python. The ROM we customized is Cyanogenmod 11 (based on Android 4.4.4). We ran experiments on an 8-processor 32GB-RAM machine with Ubuntu 14.04, OpenJDK 1.7.0_79 and Python 2.7.9. The Android device we used to run dynamic analysis is Nexus 5 Hammerhead.

The apps we selected are popular apps from Google Play downloaded in November 2015. We selected four categories to perform our experiments, including *Tools*, *Weather*, *Transportation* and *News & Magazines*. The apps are ranked in descending order by the number of reviews, and we chose apps starting from the most reviewed ones. After removing the incompatible apps that could not run on our device and the out-of-date apps that are no longer supported by developers, we use 50 apps in each of the four categories, respectively.

Applicability and Scalability

The scalability of our work depends on the automated dynamic analysis and static analysis used in PERUIM. We analyzed the apps in four categories and found that PERUIM successfully generated UI-based permission descriptions for more than 80% of them. The failures are due to multiple reasons in dynamic analysis and static analysis, as shown in Table 2.

The main restriction of PERUIM is automated dynamic analysis, where more than 16% of our tested apps failed. We manually checked the reasons of failure, and found that most of them are due to *special inputs*, because dynamic analysis cannot bypass some states of app that need special user input (login screen for instance). To solve these issues, we may need to run dynamic analysis manually as there are no mature automated solutions to deal with login screens. However, we leave this as future work as these failures are not a significant limitation of PERUIM.

Static analysis does not affect the applicability significantly, as only 1.5% of our apps failed in static analysis (mostly due to obfuscation). Even when it fails in static analysis, PERUIM is still able to finish, although giving a less complete result.

We also evaluated the time overhead of PERUIM. The time spent on dynamic analysis is under the control of the analyzer (i.e. DroidBot). One may spend more time on dynamic analysis or even test the app manually if he or she wants a higher coverage. In our experiment, we limit the automated dynamic analysis time to 5 minutes.

The time overhead of static analysis is about 23 seconds on average, which is acceptable for large-scale analysis, given the fact that highly precise static analysis tools such as FlowDroid [1] are often computationally- and memory-intensive. We made some simplifications to improve scalability, such as

skipping alias analysis and only considering the methods related to UI. Such simplifications do not significantly affect the result of static analysis, but speed up our analysis significantly.

Permission Coverage

We use this experiment to see how many permissions could be explained by PERUIM. We take the permissions listed in the manifest file as all permissions we need to explain, and we regard the permissions mapped to UI components as the explained ones. The coverage of UI-based permission description is calculated by:

$$Coverage = \frac{\#\, Explained\ permissions}{\#\, All\ requested\ permissions}$$

We found that PERUIM achieves a 61% coverage on average, and around 20% of the apps got a higher-than-90% coverage. We manually checked the apps with a low coverage and found that the main reasons include:

- The app requests more permissions than it uses, which is due to mistakes made by developers. Android apps declare requested permissions in the manifest file, while some developers just copy and paste the configurations from other apps regardless of the least permission set of their apps.
- Some functionalities cannot be reached in dynamic analysis, the automated test input generation tool did not reach some sensitive UI states thus the permissions are not triggered. The coverages of these apps could be further improved using a manual dynamic analysis.

Accuracy

We then evaluate the accuracy of PERUIM on mapping permissions to UI components.

We selected 10 popular apps from the four categories, and use PERUIM to generate permission-UI mappings for 5 screenshots in each app (Note that similar UIs are skipped). Then we manually check the results and label the correct mappings. The overall precision of permission-UI mapping is shown in Table 3, and the detailed number of mappings and number of correct mappings for the 10 apps are shown in Table 4.

We can see the average precision of PERUIM is 76.75%. However, there are also many states that only have a precision of around 50%. The reason is mainly due to the conservative approach used in static analysis when generating the *accessed* sets. For example, an app accessed a set of permissions (a *Location* permission for instance) and entered a new UI state, which contains multiple dynamic UI components (such as a "News" view and an advertisement view), then PERUIM will assume all of the dynamic UI components have accessed the permissions (both the "News" view and the advertisement view have accessed *Location* permission in our example), which is clearly a conservative result.

When comparing the results on the *will access* sets and the *accessed* set respectively, we calculated the precision of mappings on both *will access* sets and *accessed* sets and the result is shown in Table 3. The overall precision on two sets indicate that PERUIM gains a higher precision on *will access* sets, which is about 87%.

Table 3. The overall precision of Permission-UI mapping.

	will access set	*accessed* set	overall
# mapping	141	74	215
# correct	123	42	165
precision	87.23%	56.77%	76.75%

Note that we did not calculate the recall for the precision-UI mappings, because it is hard to calculate how many mapping relations are missed by PERUIM as there is no ground truth available. Furthermore, we can improve the mapping coverage of PERUIM by extending the duration of dynamic analysis or even doing dynamic analysis manually. We will investigate this direction in our future work.

FUTURE WORK

User Study

The goal of PERUIM is to help users better understand the permission requests of apps, thus we plan to add a user study to evaluate the effectiveness of PERUIM in explaining permissions to users. We plan to select UI-based permission descriptions generated from popular apps and recruit Android users to make permission access control choices with or without the UI-based permission descriptions. The effectiveness of PERUIM could be examined based on whether the users will make different (and better) choice after reading our UI-based permission descriptions.

Abnormal Behaviors

Some of the UI-based permission descriptions are interesting and even surprising. For example, a memory-clean app requested the permission to access location when it was cleaning device memory. We plan to check these abnormal behaviors to see if there are any security concerns.

UI-based Permission Access Control

With the permission-UI mappings, we are able to provide support for UI-based permission access control. It might be attractive and useful if users could select different permissions for different UI components. The challenges may include isolating UI components, providing easy-to-use configuration interfaces and supporting legacy apps.

CONCLUDING REMARKS

In order to help users better understand how and why permissions are used within a mobile application, this paper introduces *permission-UI mapping* as a fine-grained and easy-to-understand representation on how permissions are used by each UI components in an app. We have designed and implemented PERUIM to automatically extract permission-UI mappings from Android apps and visualize the results to help user understanding. We have evaluated the accuracy and applicability of PERUIM with experiments on popular Android apps.

ACKNOWLEDGMENTS

We would like to thank Prof. Jason Hong for inspirational discussions and Guofeng Shen for implementing the mobile visualization app and helping conducting experiments. This work is partly supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A203 and the National Natural Science Foundation of China under Grant No.61421091, 61370020.

Table 4. The results of Permission-UI mapping.

App	com.devexpert.weather					com.aws.android				
UI state										
# UI components	7	5	4	6	3	3	10	9	2	8
# Perm-UI mappings	16	5	4	14	8	5	6	8	2	4
# Correct mappings	12	4	4	10	6	4	4	4	2	3
App	com.avast.android.cleaner					com.ljmobile.move.app				
UI state										
# UI components	2	3	5	5	6	11	2	1	5	4
# Perm-UI mappings	1	5	2	3	4	2	1	1	1	0
# Correct mappings	1	4	2	2	2	2	1	1	1	0
App	com.magnifis.parking					nexti.android.bustaipei				
UI state										
# UI components	3	4	10	1	5	12	3	5	2	3
# Perm-UI mappings	5	6	2	4	8	14	6	1	0	0
# Correct mappings	5	6	2	2	6	13	3	1	0	0
App	com.farproc.wifi.analyzer					com.acmeaom.android.myradar				
UI state										
# UI components	3	4	6	2	3	3	5	11	2	7
# Perm-UI mappings	1	5	2	1	2	1	3	10	0	0
# Correct mappings	1	4	2	1	2	1	3	10	0	0
App	bbc.mobile.news.ww					com.mobstac.thehindu				
UI state										
# UI components	3	3	3	3	4	1	8	4	7	2
# Perm-UI mappings	2	2	2	1	0	0	10	13	13	10
# Correct mappings	2	1	1	1	0	0	6	9	8	6

REFERENCES

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. DOI: http://dx.doi.org/10.1145/2594291.2594299

2. Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 217–228.

3. Alexandre Bartel, John Klein, Martin Monperrus, and Yves Le Traon. 2014. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering* 40, 6 (2014), 617–632.

4. Android Developers. 2016a. Android Debug Bridge. http://developer.android.com/tools/help/adb.html. (2016). Accessed: 2016-03-10.

5. Android Developers. 2016b. Optimizing Your UI. http://developer.android.com/tools/debugging/debugging-ui.html. (2016). Accessed: 2016-03-10.

6. Android Developers. 2016c. Requesting Permissions at Run Time. http://developer.android.com/intl/zh-cn/training/permissions/requesting.html. (2016). Accessed: 2016-03-10.

7. Android Developers. 2016d. UI overview in Android. http://developer.android.com/guide/topics/ui/overview.html. (2016). Accessed: 2016-03-10.

8. William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

9. Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.

10. Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 3.

11. Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*. 977–992.

12. Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1036–1046. DOI: http://dx.doi.org/10.1145/2568225.2568301

13. R Uday Kiran, Haichuan Shang, Masashi Toyoda, and Masaru Kitsuregawa. 2015. Discovering Recurring Patterns in Time Series.. In *EDBT*. 97–108.

14. Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 501–510.

15. Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I Hong. 2014. Modeling usersâĂŹ mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium On Usable Privacy and Security (SOUPS 2014)*. 199–212.

16. lynnlyc. 2016. DroidBot: Automatic testing of apps in DroidBox. https://github.com/lynnlyc/droidbot. (2016). Accessed: 2016-03-10.

17. Sheng Ma and Joseph L Hellerstein. 2001. Mining partially periodic event patterns with unknown periods. In *Proceedings. 17th International Conference on Data Engineering, 2001*. IEEE, 205–214.

18. Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*. 993–1008.

19. Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications.. In *USENIX Security*, Vol. 13.

20. Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 71–72.

21. Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1354–1365.

22. Franziska Roesner, James Fogarty, and Tadayoshi Kohno. 2012a. User interface toolkit mechanisms for securing interface elements. In *Proceedings of the 25th annual*

ACM symposium on User interface software and technology. ACM, 239–250.

23. Franziska Roesner and Tadayoshi Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond.. In *USENIX Security*. 97–112.

24. Franziska Roesner, Tohru Kohno, Alexander Moshchuk, Bryan Parno, Harry Jiannan Wang, and Crispin Cowan. 2012b. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and privacy (S&P)*. IEEE, 224–238.

25. Julia Rubin, Michael I. Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert Communication in Mobile Applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

26. Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. (2016), 21–24.

27. Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. Adsplit: Separating smartphone advertising from applications. In *USENIX Security*. 553–567.

28. StevenArzt. 2016. Soot: A framework for analyzing and transforming Java and Android Applications. `http://sable.github.io/soot/`. (2016). Accessed: 2016-03-10.

29. Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 165–176.

30. Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 1107–1118.

31. Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014. Compac: Enforce component-level access control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 25–36.

32. Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android permissions remystified: a field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 499–514.

33. Wikipedia. 2016a. App Store. `https://en.wikipedia.org/wiki/App_Store_(iOS)`. (2016). Accessed: 2016-03-10.

34. Wikipedia. 2016b. Google Play. `https://en.wikipedia.org/wiki/Google_Play`. (2016). Accessed: 2016-03-10.

35. Wikipedia. 2016c. User interface. `https://en.wikipedia.org/wiki/User_interface`. (2016). Accessed: 2016-03-10.

36. Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. 2013. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1043–1054.

37. Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 518–529.

38. Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 9–18.