UNDERSTANDING USER INTERFACE FOR INTELLIGENT SOFTWARE ANALYSIS AND EXPLORATION

by

XUE QIN, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Xiaoyin Wang, Ph.D., Co-Chair
Jianwei Niu, Ph.D. , Co-Chair
Wei Wang, Ph.D.
Palden Lama, Ph.D.
Travis Breaux, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2020

DEDICATION

I would like to dedicate this thesis/dissertation to my family.

ACKNOWLEDGEMENTS

First, I would like to thank the financial assistance from NSF and the University of Texas at San Antonio during my Ph.D. study. Second, I would like to thank my supervisor, Xiaoyin Wang, for his kindness and selflessness. He guided me through many difficulties from life and work. And he inspired me on how to be an excellent educator, a helpful friend, and a supported teammate. I want to acknowledge Professors Jianwei Niu and Travis Breaux, for inspiring my interest in the development of innovative technologies. My research partners, Mitra, Rockey, John, and Xueling, thank you for all the kind help, and I am grateful for our valuable collaborations. I wish we could continue in the future. Last, I would like to thank my family for supporting me in these years. Without you, I cannot finish this task.

May 2020

UNDERSTANDING USER INTERFACE FOR INTELLIGENT SOFTWARE ANALYSIS AND EXPLORATION

Xue Qin, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professors: Xiaoyin Wang, Ph.D. and Jianwei Niu, Ph.D.

Nowadays, millions of people across more than 190 countries around the world are using smartphones every day. People use them to book hotels, online shopping, and keep track of fitness information. The Graphic User Interface (GUI) is an essential part of an app to communicate with users. GUI analysis will help improve the security, privacy, and reliability of the applications and prevent the potential loss for both app companies and end-users. Existing research works on mobile apps are mainly focusing on system-defined APIs. Unlike well-defined APIs, GUIs are more unstructured and implicit since there is no criterion. Thus it makes analyzing and exploration really hard. To overcome such challenges, we proposed a novel approach to UI analysis and exploration. We applied it to solve the problems in privacy security and GUI testing reusing in mobile applications. We found 39 violations out of 120 apps in privacy analysis, and successfully convert averagely 80.2% of the UI test cases from iOS to Android.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Problem Statement

There are around 2.5 billion smartphone users across the world by 2019. And 3.6 million apps in Google Play store until March 2018. Graphic User Interface (GUI) is one of the essential parts for an application or graphic software to communicate with users. It collects users' input, e.g., text input, voice input, and provide customized services based on users' requirement. A Better GUI will enhance the user experience.

GUI is also one of the significant components to reflect the behaviors of an application or a system. By analyzing or understanding the behavior, developers can improve the reliability, privacy security of the application and can also reveal the potential security defects.

But GUI analysis and exploration is not easy. One of the primary reasons is that the machine cannot understand the UI like human beings. Unlike well-defined APIs, GUIs are more unstructured and implicit since there is no criterion. We list three challenges in analyzing GUIs and proposed possible solutions in the following sections.

1.1.1 Hierarchical Information Type

As we mentioned above, GUI is a way that allows an application to communicate with users. The process of communication includes information exchange. That is to say, the application tells users what data it is collecting, and responses with the information that users expected. Thus, each exchanged data should come with an information type that can be understood by both ends. For example, in the shipping page from a shopping application, there must exist input fields to let the user provide the delivery location. The information types related to those inputs are shipping address.

And most of the time, the information type for one input field needs to be understood with the help of the context and layout. We refer this information type to be a hierarchical-information-type.

Figure 1.1 shows an example of this information type. We currently decide the information

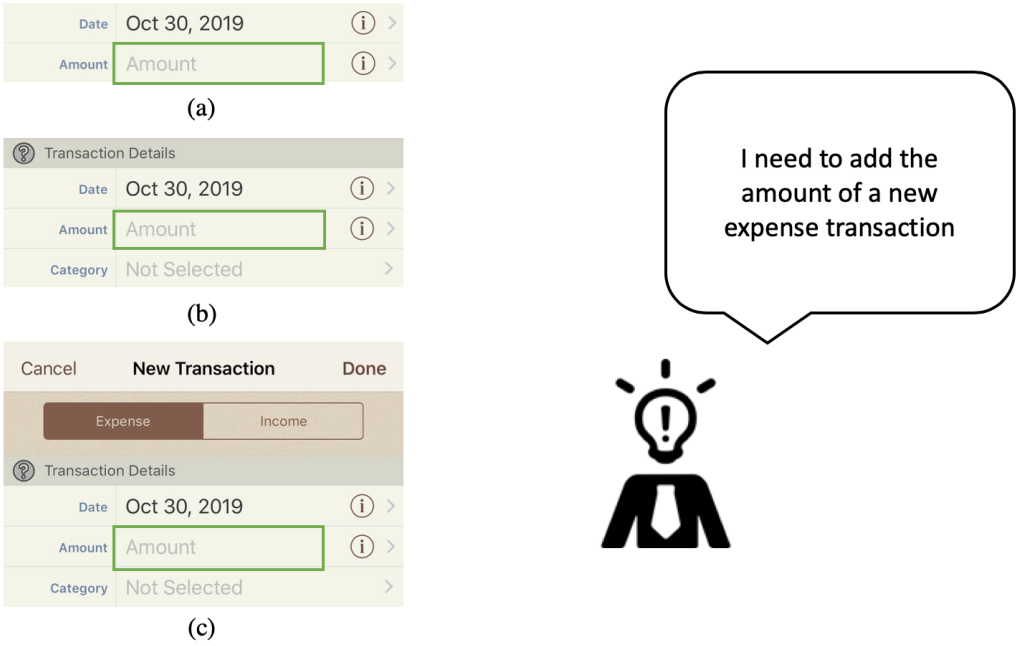


Figure 1.1: Hierarchical Information Type

type of the input box highlighted with a green rectangle. (a) is a partial screenshot of an input box with label Amount and Date. By only viewing the "Amount," it is impossible to infer what amount the app is asking to input. It could be the amount of intake water from the user, or it may be the amount of deposit the user is saving. Then, we add a few details in (b). From (b), we know the app is asking the user to input the amount of a transaction on Oct 30th, 2019. But we still don't know what transaction has been asked. The full screenshot is shown in (c) and from which we have a clear picture that the user is asked to add the amount of a new expense transaction on Oct 30th, 2019.

This example shows us how the developers deliver the message precisely to app users by adding more detailed information on the GUI structure. In other words, without the acknowledgment of the context, it is difficult to find out the information type directly.

1.1.2 Varies of GUI Design

In the last section, we discuss the hierarchical information type. Another challenge based on this is the variety of the GUI design, which makes GUI analysis and exploration face numerous difficulties.

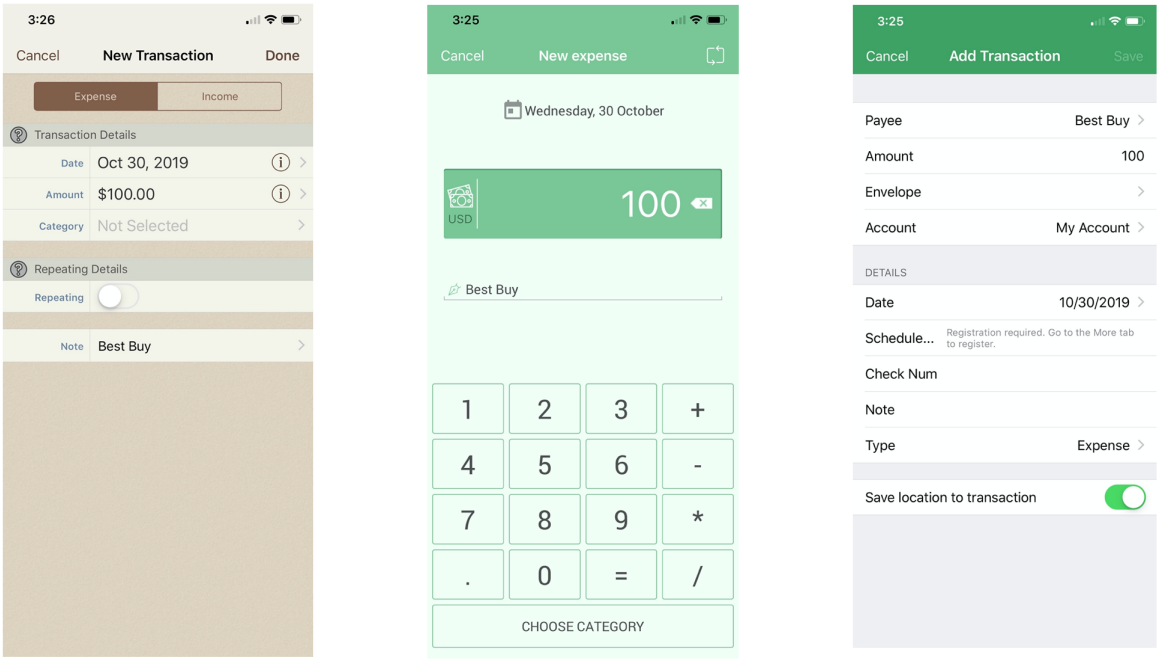
Every developer has its own way to design GUI. Moreover, when it comes to business apps, companies like to keep their apps in unique styles. All these factors make GUIs more unstructured and implicit since there is no criterion. And as more and more apps have been brought up to the market, the testing and analyzing works seem unbearable.

Figure 1.2 displays screenshots from three personal financial Android applications: Spending, Monefy, and Goodbudget. They are all popular apps to help the user manage their budgets and record every transaction. In the three screenshots, all of them are adding a new $100 expense transaction in Best Buy. As human beings, it is not hard for us to understand the information types that GUIs are describing. But it will be a difficulty for the machine to understand these three screens are doing the same thing. And the more complex the design or the structure is, the harder can analyzing tool understands it.

1.1.3 Constraints among GUIs

Other than the challenges of the individual GUI analysis, there are also many difficulties lying in the multiple GUIs analysis. As we know, each GUI in the application or system is not isolated, and it also connects with other GUIs to serve specific functional requirements for the system. In other words, there exist logical relationships between multiple GUIs, and most of them depend on each other. The logic within the GUIs can be easily found in the application's source code and can be understood by human beings. However, it is difficult for the machine to infer these types of connections, especially without the source code, which is a common scenario in GUI testing.

Figure 1.3 shows a sequence of screenshots from Amazon Android application where the user is trying to purchase a product. The first screenshot from the left is the information page of the product. After the user clicked the "Add to Cart" button, the app directs the user to the page of a



Spending Monefy Goodbudget

Figure 1.2: Varies of GUI Design

shopping cart. And after clicking the "Proceed to checkout" button, the user finally jumps to the order information page and tap the "Place your order" to finish the purchase.

There are at least three hidden constraints in the procedure above. First, users have to add the product to the shopping cart before making a purchase. The items in the shopping cart depending on the operations on the product page. Second, An empty shopping cart cannot proceed to check-out. This logic is straightforward for the user to understand since it is a common sense and can be very straightforward to implement in source code, for example, an if condition. But it can be tough to let the machine remember all these "common senses". Last, before the user clicking the "Place your order" button, we assume he has already set up the shipping address and payment method. Otherwise, the user will be direct to these two pages to filling the information first. These are just a few examples and we could name a lot more.

The challenge is, how can we reveal and include these constraints in the GUI analysis or exploration to achieve our goal such as better testing coverage?

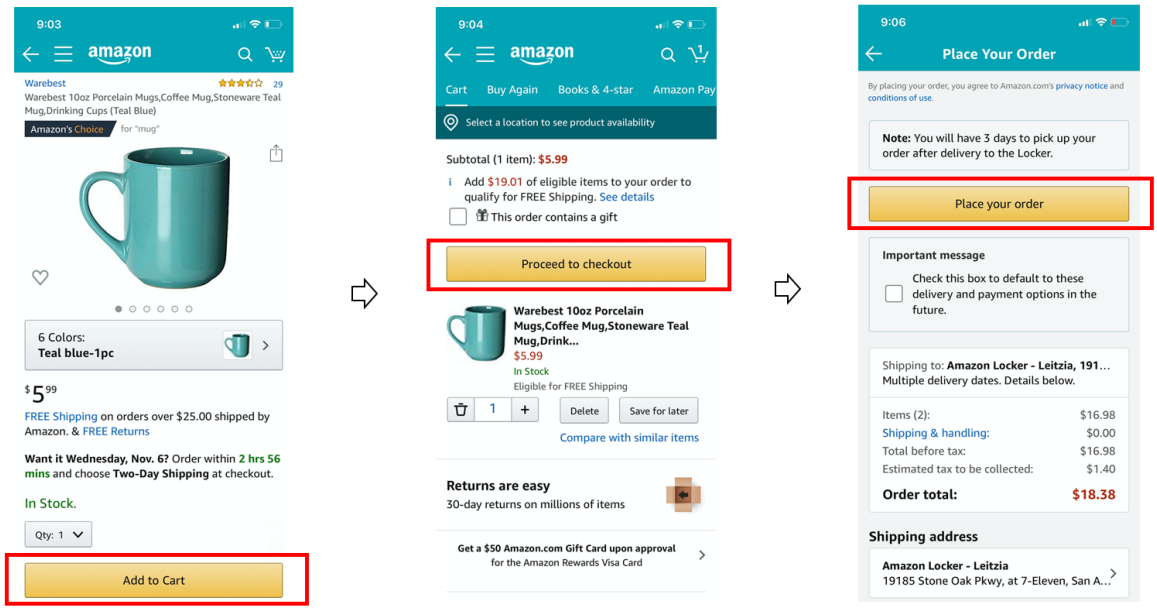


Figure 1.3: Hidden constraints between GUIs

1.2 Thesis Statement

To narrow down my research work, I currently focus on mobile applications. And my thesis statement is below:

> ***How much GUI understanding can enhance the analysis and exploration in the mobile app?***

1.3 Contribution

In this section, I briefly discuss the research results we have done to solve the three challenges.

Before solving the first challenge, we conducted a user study [180] to further understand how humans read the GUI labels, and we let 24 participants perform tasks on Android apps with different GUI quality levels. And we found out 32% of the users correct their misunderstandings from the help of the GUI context. And the GUI context often includes text labels, icons, color patterns, images, and layout. We focused on text labels and tried to solve the three challenges.

To solve the first and second challenge problem, we propose a novel approach [221] that analyzing the hierarchical information for each screen instead of the individual label for certain GUI elements. When mapping the GUI text label to information type, we propose a hierarchical mapping to take the context into consideration. We further applied this technique to solve the privacy security issue, which is detecting inconsistencies between the app's code, collection behavior of user input data, and collection statements in mobile app privacy policies. Using crowdsourcing tools, we developed domain-specific privacy ontology for three privacy-sensitive domains: health, finance, and dating. By mapping the information type in the ontology and the hierarchical information type on GUI, we detected 39 violations in total on 120 most popular apps in the health, finance, and dating categories.

For the third challenge problem, although we could not directly infer the logical relationship purely base on GUI analysis, we propose a framework [181] that could reuse the existing relationship from current mobile applications and apply the comparable exploration on a similar application. More specifically, we proposed a novel approach to automate the migration from iOS GUI tests to Android GUI tests. And we evaluate five popular open-source mobile applications that have both iOS and Android versions. The results show that our approach can successfully migrate 80.2% of the recorded iOS GUI events to Android GUI events on average, and averagely our migrated test cases achieved a similar test coverage as the original test cases (59.7% vs. 60.4%)

1.4 The structure of the Thesis

The remaining of this thesis is organized as follows. In Chapter 2, we present the background knowledge of all the concepts we touched in this thesis, including what is GUI, GUI structure, and GUI testing. Then we introduce the related work in the challenges we are trying to solve in Chapter 3. In Chapter 4 and Chapter 5, we present the details of our approaches and the evaluations to solve the challenges listed at the beginning of this chapter, and in Chapter 6 I briefly present the future directions and the preliminary work I have done. Final conclusion is in Chapter 7.

CHAPTER 2: BACKGROUND

In this chapter, we first demonstrate the Graphic User Interface (GUI) in the mobile application, including how it constructed and what are the difficulties in analyzing. Then we will briefly introduce what GUI testing is and what are the conventional techniques. At the very end, we will discuss the commonalities and differences in the GUI structures and GUI tests between Android and iOS platforms.

2.1 Graphic User Interface in Android application

Graphical User Interface is a user interface that includes graphical elements, such as windows, icons, and buttons. This term was created in the 1970s to distinguish graphical interfaces from text-based ones, such as command-line interfaces. However, today nearly all digital interfaces are GUIs. Other than using a mouse and a keyboard. A mobile operating system such as Android is designed to use a touchscreen interface and take the input, such as tap, swipe, and scroll.

In Android, a user interface is defined using the layout. All elements in the layout are built using a hierarchy of *View* and *ViewGroup* objects. A View usually draws something the user can see and interact with. Whereas a *ViewGroup* is an invisible container that defines the layout structure for View and other ViewGroup objects, as shown in figure 2.1.

The *View* objects are usually called "widgets" and can be one of many subclasses, such as *Button* or *TextView*. We can think of the widget as the basic component of the entire screen. The

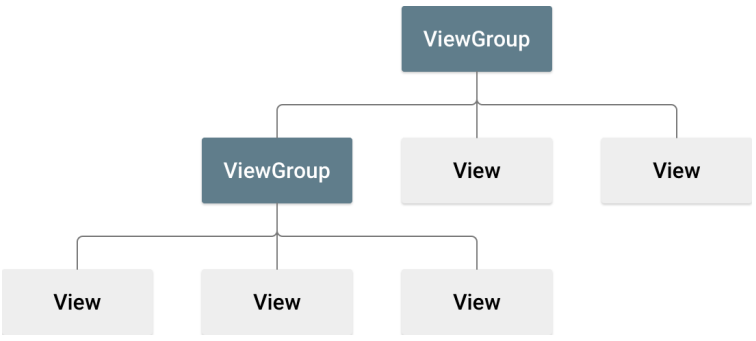


Figure 2.1: Android View Hierarchy

ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as *LinearLayout* or *ConstraintLayout*. These layouts define where and how we should place each GUI widget.

We usually have two ways to declare a layout:

- **Declare GUI layout statically.** The easiest way to statically declare the GUI layout is to declare them in XML. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts. Developers can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.

- **Declare GUI layout dynamically.** Declaring the GUI layout dynamic is to instantiate layout elements at runtime. In other words, the layout information will be generated programmatically as you encounter it when exploring or using the app, and this information cannot be found in the resource files like XML.

The major challenge in building the hierarchical information type is to precisely capture the layout structure, including both static and dynamic declaration. And it is not easy to directly get the dynamic declaration since we usually don't have access to source code. In the next section, we will introduce the dynamic layout in detail.

2.2 Dynamic Layout

As we discussed in the last section, A *layout* in the Android framework defines the visual structure of the GUI, including locations for views, buttons, windows, and widgets. Layouts are defined in two ways: either using XML to define the placements of elements statically or creating programmatically at runtime. The Dynamic declaration is necessary and has been widely applied in real-world applications when layouts need to be dynamically changed based on runtime states. These *dynamic layouts* eliminate the need to pre-draw multiple GUIs. These two methods are

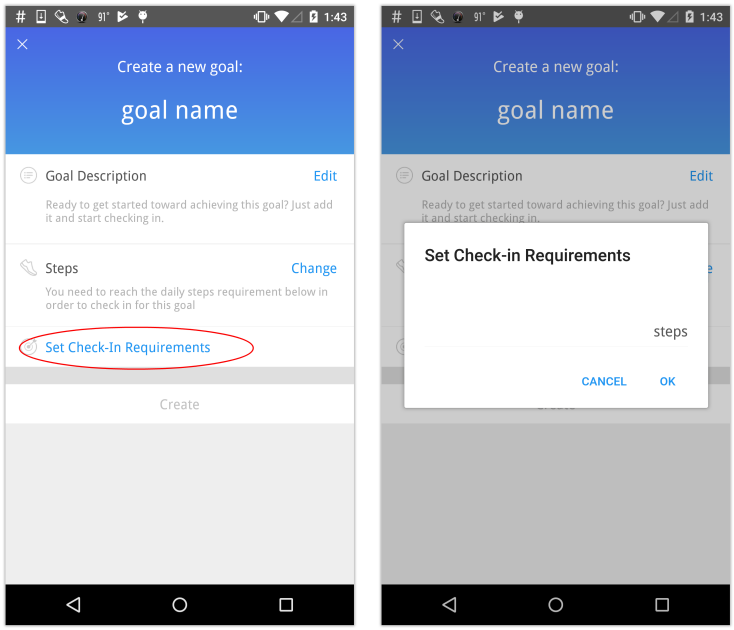


Figure 2.2: Screenshots from Pacer

combined flexibly in the application. For example, a label or View can be programmatically added to a statically defined layout (typically after inflation).

The example in Figure 2.2 shows two GUI screenshots from Pacer, a popular fitness app, depicting the GUI when a user creates a new exercise goal. Besides editing goal descriptions and changing goal types such as steps and diet, the user also needs to set the check-in requirement by clicking the button in the red oval on the left screenshot. The right screenshot shows the pop-up window that appears after clicking this button. Here, users will be asked to type in the desired number of check-in steps.

The right screenshot utilizes a combination of both static and dynamic layouts. Listing 2.1 shows the static definition of the right screenshot, which is a *layout template* defining the basic layout structure and the font/style information. All the labels and IDs are vaguely defined (e.g., Dialog title is undefined, and "et_content" is used as the view id of the input box). Thus this layout template can be used in multiple places in the project for user input, and the labels (e.g., Set Check-In Requirements for title) will be transferred from the parent activity (e.g., the activity in the left screenshot) when the dialog is opened.

Listing 2.2 shows the smali code (decoded Android bytecode) in `InputDialogFragment.smali` which dynamically adds the label for "Set Check-In Requirement". The string is fetched at Line 1 as `v2` with the id `0x7f07011a`. Here, the id references the appropriate string in `string.xml` based on the context. In Line 2, `v2` is passed as a title resulting in "Set Check-In Requirement" being dynamically defined as the title.

```
1 <LinearLayout android:gravity="center_horizontak" ...>
2   <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView
3     ... android:id="@id/title" ... />
4   <cc.pacer.androidapp.ui.common.fonts.TypefacedEditText
5     ... android:id="@id/et_content" ... />
6   <LinearLayout ...>
7     <Button ...
8       android:id="@id/btnLeft" ...
9       android:text="@string/btn_cancel" ... />
10  </LinearLayout>
11 </LinearLayout>
```

Listing 2.1: Partial Code from `common_input_dialog.xml`

```
1 const  v2, 0x7f07011a
2 invoke-virtual {v1,v2}, Lcom/afollestad/materialdialogs/MaterialDialog\$
    Builder;->title(I)Lcom/afollestad/materialdialogs/MaterialDialog\$Builder;
3 move-result-object v1
```

Listing 2.2: Virtual Invoke Example in `GoalSetCheckingInReqDialog.smali`

```
1 <LinearLayout android:gravity="center_horizontak" ...>
2 <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView ...
3   android:text="@string/goal_set_requirements" ... />
```

Listing 2.3: Partial Code from `goal_create_details_fragment.xml`

2.3 Hierarchical Information with GUI Context

As we discussed in Section 1.1, the information type in GUI is usually hierarchical. In other words, the individual GUI element text label needs to be understood with its GUI context. This is just like how we understand one unseen word when reading a paragraph. We often find the answer by observing the context around this word: other words around it and maybe sentences around it. And we do the exact same thing in GUI analysis: understanding the individual GUI widget by observing the context.

So just like the contexts in natural language paragraphs, input views can only be well understood with neighboring ancestor views. In the right screenshot, without seeing the title, it is difficult to understand what is supposed to be entered into the field. Furthermore, the left screenshot that leads to the right dialog also provides context information for the dialog. This invoking view can be found in the resource layout file `goal_create_details_fragment.xml`, as shown in Listing 2.3, and the view's label referring to `@String/goal_set_requirements`.

GUI context is essential in understanding user input views and mapping the views to privacy-policy phrases, but the dynamic implementation of Android GUI makes identification of the GUI context difficult. In this paper, we propose input context analysis to handle dynamic implementation and hierarchical mapping to map input views to privacy policy phrases based on collected GUI context.

2.4 GUI Layout in Android and iOS

As the two most popular mobile platforms, the UI frameworks of Android and iOS share lots of common features. First, although referred to with different names, both UI frameworks have a three-level GUI hierarchy: a screen containing various GUI controls that users can interact with (called a scene in iOS, and an activity in Android), a group of GUI controls forming a functional area in the window (called a `UIView` in iOS, and a `ViewGroup` in Android), a basic GUI control (called a `UIControl` in iOS, and a `View` in Android). Here, in the framework design, the

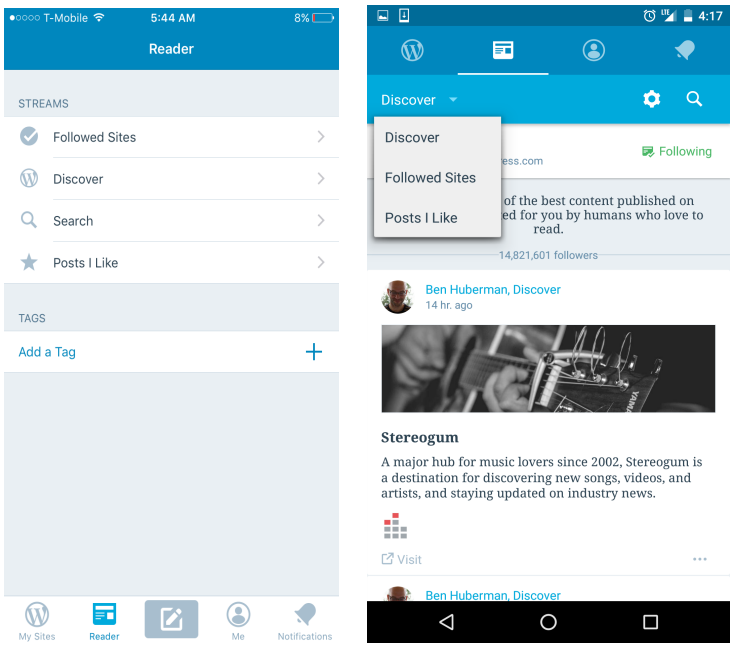


Figure 2.3: GUI difference between iOS and Android

two lower levels of UI elements are instances of the same abstract class, *i.e.*, `UIControl` is a subclass of `UIView` in iOS, and `ViewGroup` is a subclass of `View` in Android. Second, iOS and Android UI frameworks share a similar group of basic GUI controls (*e.g.*, check boxes in Android and switches in iOS), and mobile style UI controls (*e.g.*, date pickers, and sliders). Although the naming is slightly different, it is feasible to construct the mappings between the GUI control types of two GUI frameworks.

Despite the commonality in the high level GUI model, there are numerous differences between iOS and Android GUI frameworks. We summarize as follows the major ones that may affect the task of GUI test migration. First, Android supports the global return key, and supports a context menu once with the global menu key and now with the menu key in the action bar. By contrast, these keys are not supported in iOS, so iOS apps tend to define and use their own UI controls to navigate backward and design their own UI views for context menus. Second, iOS apps and Android apps have different UI design styles. For example, while iOS apps typically use tabs at the bottom of the screen to switch between different views in one screen, Android apps often use a drop-down menu at the left or right top corner. Note that, developers have the motivation to adapt

```
1       let app = XCUIApplication()
2       let mainNavigationTabBar = app.tabBars["Main Navigation"]
3       simpleLogin(username: WordPressTestCredentials.oneStepUser,
4       password: WordPressTestCredentials.oneStepPassword)
5       self.waitForElementToAppear(element: mainNavigationTabBar)
6       mainNavigationTabBar.buttons["My Sites"].tap()
7
```

Listing 2.4: Sample iOS GUI Test

```
1   onView(withId(R.id.greet_button))
2   .perform(click())
3   .check(matches(not(isEnabled())));
4
```

Listing 2.5: Sample Android GUI Test

their UI design to the style of the target platform, because otherwise, their app may not fit well with the user's habit in that platform.

Figure 2.3 shows two screen shots from the iOS version (left) and the Android version (right) of WordPress. The two screens are both reached by starting the app, and clicking on the "reader" button (the second icon from left in the bottom tabs of the iOS version, and the second icon from left in the top tabs of the Android version). However, they look very different. For the reading feature, the iOS version organizes the four sub-features ("followed sites", "discover", "search", "posts I like") in a list of views, and the user can click on the specific item in the list to reach the sub-feature. By contrast, the Android version directly shows the sub-feature of discover, and the user can switch to other sub-features by clicking on the items in the drop-down list (top-left corner of the screen). Furthermore, the iOS version has a "Add a Tag" feature which is not supported by the Android version.

As a result, if an iOS user wants to reach the "discover" sub-feature, she needs to trigger two events: clicking on the "reader" icon, and clicking on the list item "Discover". But an Android user needs to trigger only one event: clicking on the "reader" icon. However, for the other three features ("followed sites", "search", and "posts I like "), an iOS user still just needs to trigger two events, but an Android user needs to trigger three events: clicking on the "reader" icon, clicking on the drop-down menu in the top-left, and click on the specific menu item. From the two screen

shots, we have the following observations.

- iOS and Android versions of apps have similar features and sub-features, and they organize features in a similar way. Therefore, it is feasible to transform an iOS event trace to an Android event sequence.
- The GUI views/controls correspond to the same feature/sub-feature are very similar. For example, the "reader" icons in both versions look the same, and the list items/menu items corresponding to the four sub-features also have the same label.
- Due to various issues, to fulfill a certain task, it can take different steps in one version compared to the other. Therefore, although the mappings of the GUI views and controls are often one-to-one, the mappings between GUI events are typically many-to-many.

Based on the above observations, TestMig records the event trace in the iOS version, construct mappings between GUI controls, but uses sequence transduction to construct the many-to-many mapping on GUI events.

2.5 GUI Testing and Exploration

In the above three sections, we introduce what GUI and the way to compose it is. These fundamental understanding will help us in GUI Testing and Exploration. And in this section, we will briefly introduce what GUI Testing is, why we need it, and the standard approaches.

2.5.1 What is GUI Testing?

GUI Testing is a software testing type that checks the Graphical User Interface of the Application Under Test. GUI testing involves checking the screens with the controls like menus, buttons, icons, and all types of bars - toolbar, menu bar, dialog boxes, and windows, etc. The purpose of Graphical User Interface (GUI) Testing is to ensure GUI functionality works as per the specification.

GUI Testing usually tests the various aspects of the user interface, such as:

- Visual Design
- Functionality
- Security
- Compliance
- Usability
- Performance

2.5.2 Why is GUI testing important?

Modern applications are beyond the desktops. They are either mobile-based or cloud-based applications. They need to be more user-friendly as per customer demand. The application interface and user experience play a significant role in application success as it is released to the market. A GUI testing team always pays close attention to each detail in visual dynamics to ensure end-user satisfaction and ease. GUI Testing will help to releases less error application software, and it will increase the efficiency of software and improve software quality.

2.5.3 What are GUI Testing Approaches?

There are generally three approaches using in the industry:

- **Scripted testing.** In scripted testing, software testers design and then execute pre-planned scripts to uncover defects and verify that an application does what it is supposed to do. For example, a script might direct a tester through the process of placing a specific order on an online shopping site. The script defines the entries that the tester makes on each screen and the expected outcome of each entry. The tester analyzes the results and reports any defects that are found to the development team. Scripted testing may be performed manually or supported by test automation.

- **Exploratory testing.** Rather than following pre-written test scripts, exploratory testers draw on their knowledge and experience to learn about the AUT, design tests, and then immediately execute the tests. After analyzing the results, testers may identify additional tests to be performed and/or provide feedback to developers. Although exploratory testing does not use detailed test scripts, there is still pre-planning. For example, in session-based exploratory testing, testers create a document called a test charter to set goals for the planned tests and set a time frame for a period of focused exploratory testing. Sessions of exploratory testing are documented by a session report and reviewed in a follow-up debriefing meeting. Likewise, during scripted testing, there may be some decisions available to testers, including the ability to create new tests on the fly. For that reason, it is helpful to think of scripted testing and exploratory testing as being two ends of a continuum rather than being polar opposites. Both scripted and exploratory testing can be completely manual, or they can be assisted by automation. For example, an exploratory tester might decide to use test automation to conduct a series of tests over a range of data values.

- **User experience testing.** In user experience testing, actual end-users or user representatives evaluate an application for its ease of use, visual appeal, and ability to meet their needs. The results of testing may be gathered by real-time observations of users as they explore the application on-site. Increasingly, this type of testing is done virtually using a cloud-based platform. As an alternative, project teams can do beta testing, where a complete or nearly-complete application is made available for ad hoc testing by end-users at their location, with responses gathered by feedback forms. By its nature, user experience testing is manual and exploratory.

CHAPTER 3: RELATED WORK

In this chapter we discuss the related work in GUI analysis and exploration, as well as the privacy concern practice.

3.1 GUI analysis and Privacy Policies

To our knowledge, ours is the first approach that uses data flow analysis to verify consistency between app-collected data and privacy policy language with regard to native code and user input. The following are related works in the area of Android data flow analysis and privacy policies.

Slavin et al. [198] and Zimmeck et al. [256] used similar approaches to detect privacy policy violations in Android apps based on Android API calls. Such an approach is useful in identifying leaks where the API calls collect personal information from a mobile device. Different from their approach, ours has the ability to detect violations based on native code. By mapping privacy-policy Phrases to user input views, we are able to go beyond Android API-based violation detection and identify potential violations involving user input. Furthermore, the API-based approach relies on developer documentation for the mapping whereas policies are not typically written by developers. For our approach, privacy-policy Phrases are mapped with a user-oriented perspective which is closer to the language of privacy policies, which we assert is more relevant since privacy policies are written with an intent to be understood by end users.

Existing work on UI-related information explore various facets of privacy and security. Huang et al. match text from UI components to top-level functions in order to detect clandestine behavior [121]. Their work targets functions by identifying suspicious permissions. In contrast, our work compared the consistency between privacy policies and user input via UI components which are based on native code. This allows our approach to not be limited by the coarse granularity of Android permissions.

Huang et al. [121] modeled so-called "stealthy behaviors" as program behavior that mismatches with user interface by using static analysis. They extracted text from the user interface component

and match them with top-level functions. Similarly, SUPOR [119] and UIPicker [164] identify sensitive input views to which sensitive information can be entered. Compared with these efforts, (1) besides detecting unusual information collections, GUILeak further checks whether the information collection is mentioned in the privacy policy based on ontology and mapping techniques to map UI views to privacy terms, and (2) on code analysis, GUILeak adapts Gator to extract contexts of input fields in dynamically generated dialogs and layouts as shown in Section **??**. In contrast, SUPOR and UIPicker extract context from only static layout xml files.

Huang et al. developed BIDTEXT as a tool for reporting the propagation of label set variables corresponding to sensitive text labels to sinks [120]. It examined text labels from either code or UI and relied on a keyword set to determine the sensitiveness of computed texts. But BIDDTEXT is not able to solve our problem because it does not try to map sensitive labels to phrases in the privacy policies. It is impossible for one keyword set to cover all types of sensitive data across apps from different categories. However the domain specific ontologies we built could solve this problem.

DESCRIBEME, developed by Zhang et al [249], is another tool that automatically generates security-centric description and bridges the gap between permissions and descriptions. It helps end users to better understand what information types have been collected through permissions. But the limitation is that app permissions cannot cover all types of collected data, especially the user input data from GUI. In our approach, we explored the privacy policies and identified all the information types that will be collected by app developers.

3.2 GUI Testing of Mobile Software

GUI Analysis. Since the emergence of graphical user interfaces, there have been a large number of research efforts focusing on the extraction of an abstract model to describe the GUI, and summarize the relations among the GUI controls [139]. In most modern GUI frameworks, such as Microsoft.Net [16], Java Swing [14], iOS Cocoa Touch [10], and Android [7], a hierarchical model is used to present the structure of GUI windows.

Generally, these models fall in two categories: structural models and behavioral models. Structural models describe the static structure of a given user interface, such as the hierarchy of containers, and the positional relationship among GUI controls. By contrast, behavioral models describe the runtime behavior of the GUI, such as the available events for users to invoke in a window, or the allowed sequence of events.

In hierarchical models, all GUI controls shown in a window are organized as a tree, in which a leaf present an elementary control (such as a button, a text area), while an intermediate node presents a composite control that consists of a number of children which can be either elementary controls or other composite controls. These hierarchical models works well for the GUI rendering and the GUI design process, and they can usually be loaded directly from the GUI-description resource files in the software project (e.g., layout.xml for Android). However, these models are not able to describe the changes made to the GUI elements at runtime.

To describe the runtime behavior of GUI, a number of models which summarize possible GUI event sequences at runtime have been studied, mainly by researchers working on GUI testing. These models may be in forms of automaton [82, 234], grammar [53], and AI Planning [143], etc. Also, a lot of techniques for the automatic extraction of these models are proposed, either based on dynamic analysis [44, 155] or static analysis [74, 204]. However, these models describes only the set of clickables to be enabled at a runtime state of a window, while the user-visible strings, as well as the structural relationships among them are out of the scope of these models. To extract the context information of user-visible strings, we need a summarization of the runtime structural relationship among all user-visible strings.

the main purpose of these models is to detect all possible event sequences to maximize test coverage. Therefore, Therefore, existing models cannot be directly applied because GUI-description files do not cover runtime changes to GUI (so a user-visible string is not covered if it became user-visible only at runtime), and event-sequence models do not provide sufficient information about user-visible strings and their relationships.

3.3 Cross-Platform Code Migration

The academia has noticed the difficulties in the library migration, and many research efforts have been made on the topic. There have been many empirical studies [182] [154] to explore the prevalence of backward incompatibility in library upgrade. Furthermore, Linares-VÃ¡squez et al. [144] studied the relationship between change proneness of APIs methods and the successfulness of client software using the methods, and found that backward incompatibilities have significant negative effect on the successfulness of client software.

The research topic most relevant to our work is support for library migration, including the mapping of APIs between two consecutive versions of a software library. Godfrey and Zou [110] proposed a number of heuristics based on text similarity, call dependency, and other code metrics, to infer evolution rules of software libraries. Later on, S. Kim et al. [131] further improved their approach to achieve fully automation. Dagenais and Robillard [90] proposed *SemDiff*, which infers rules of framework evolution via analyzing and mining the code changes in the software library itself. Wu et al. developed *AURA* [237], which further involves multiple rounds of iteration applying call-dependency and text-similarity based heuristics on the code of software library itself. More recently, HIMA [159] further enhances *AURA* by involving information from code commits between two consecutive versions of software libraries. There are also research efforts on recommending code change patterns for API migration. Nguyen et al. [166] proposed techniques to mine code-change patterns for API migrations, and to recommend code changes based on the patterns given code context.

Most existing approaches recommend API changes, instead of code changes. Meng et al. [157, 158] synthesize change scripts from code samples. Their proposed techniques may also be applied to resolving migration failures. However, their techniques require an existing fix of the migration failure, which is not always available. Furthermore, their techniques only guarantee syntactic correctness because determining semantic correctness of code edits in general scenarios is extremely hard. By contrast, TestMig achieves semantic correctness (at least to the extent supported by existing test suites) in migration-failure resolution.

CHAPTER 4: GUILEAK: A NOVEL GUI ANALYSIS WITH PRIVACY PRACTISE

This chapter has published on International Conference on Software Engineering (ICSE) 2018, and was co-authored by Xiaoyin Wang, Mitra Bokaei Hosseini, Rocky Slavin, Travis Breaux ,and Jianwei Niu.

In this chapter, we will carefully introduce the novel approach we proposed to solve the first and second challenge problem we mentioned in Chapter 1.

The Android mobile platform supports billions of devices across more than 190 countries around the world. This popularity coupled with user data collection by Android apps has made privacy protection a well-known challenge in the Android ecosystem. In practice, app producers provide privacy policies disclosing what information is collected and processed by the app. However, it is difficult to trace such claims to the corresponding app code to verify whether the implementation is consistent with the policy. Existing approaches for privacy policy alignment focus on information directly accessed through the Android platform (e.g., location and device ID), but are unable to handle user input, a major source of private information. To solve this problem, we propose a novel approach that automatically detects privacy leaks of user-entered data for a given Android app and determines whether such leakage may violate the app's privacy policy claims. For evaluation, we applied our approach to 120 popular apps from three privacy-relevant app categories: finance, health, and dating. The results show that our approach was able to detect 21 strong violations and 18 weak violations from the studied apps.

4.1 Introduction

Mobile applications (apps) are becoming increasingly pervasive. By June 2017, the Google Play Store surpassed three million apps [2], and the Android platform held 85% of the smartphone OS market share [3]. Among these apps, health and finance apps can be particularly privacy-sensitive. In 2015, health apps were downloaded by 58% of mobile phone users [137]. Such apps

collect information on body measurements, diet, exercise, and medical treatment, among others. Similarly, the 73% of the personal finance app Mint's 20 million users pay their balances every month through through the app [4].

With ease of access to personal information and the large scale of mobile app deployment, developers need better tools to help protect user privacy. Google encourages app developers to provide users with privacy policies [198], however, innovation and competition among mobile app developers can introduce inconsistencies between the application code and privacy policies. Unlike security threats where there may be malicious developers hoarding personal data, the privacy threat motivating our work is the developer who unintentionally collects personal data without informing the policy author, or where the software changes over time to yield and outdated policy.

Prior work by Slavin et al. [198] and Zimmeck et al. [256] traced privacy policy statements about the collection of platform information to application program interface (API) calls using static program analysis. These API calls concern personal information that is automatically collected from the device, such as user location, device identifiers, contact information, and sensor data. This prior work is limited, because it does not account for personal data that *users provide directly through an app's graphical user interface (GUI).* Figure **??** shows an example where sensitive data is provided to the app via the interface and is thus disconnected from any API call. In Figure **??**, the user manually enters the steps they have taken using a text field. There are many ways, both static and dynamic, to render the field and link the information provided through the field to program-level data types. Furthermore, the field itself may not have tight constraints on the input values, making it difficult to determine the information type.

These unaddressed, GUI-related challenges further widen the gap between privacy policies and app-based data practices. To bridge this gap, we identified two new technical challenges that we address in this paper:

TC1: Vague and Unbounded Information Types for User Input Data. The information types automatically collected through platform API methods are constrained to what is collectible by hardware commonly shared across mobile devices. This constraint limits the terminological

space to only a few general category names (e.g., location, voice, camera). In contrast, developers can design novel user interfaces that ask users to provide potentially *any kind of information*, which includes unstructured and semi-structured personal information in different formats and language types.

TC2: Varying GUI Implementation Techniques. Unlike platform API method calls that can be detected by scanning the app byte code, user interfaces can be implemented using static declarations in resource files or programmatically in the code. Techniques, such as SUPOR [119] and UIPicker [164], can be used to identify input views (GUI views accepting user inputs) receiving sensitive user input, but they do not map these views to relevant policy terms, nor do they identify programmatically-generated input views.

In this paper, we present a novel technique to detect privacy-policy violations on user-provided information for Android apps. The approach maps each input view to policy terminology through an ontology, and then performs static information flow analysis to detect information flows that violate relevant policy statements. To address **TC1**, for each input view in a new app, we use phrase similarity measurements to map GUI labels together with its context to ontology concepts. The ontology is then matched to the policy text. To address **TC2**, we developed a GUI string analysis technique to estimate the structure of programmatically generated GUIs and collect all GUI labels in the context of a given input view. Our analysis is based on GATOR [187], an existing GUI analysis framework for Android.

To validate our approach, we focus on three app categories in the Google Play Store: health, finance, and dating. These three categories are important because they can require access to sensitive personal information. Furthermore, the first two domains are regulated by the Health Insurance Portability and Accountability Ac (HIPAA) [185] and Gramm-Leach-Bliley Ac (GLBA) [85], respectively. In our experiment, we collected 150 of the most popular apps and their privacy policies (50 apps for each category), setting aside 20% of the apps for training. Using the privacy policies from the training apps, we constructed an ontology for each of the three domains. We then applied our approach to the remaining 120 apps and detected 39 violations, which we manually confirmed

by recording the runtime network requests with the Xposed framework[1].

4.2 Approach

The overview of GUILeak is depicted as a data flow diagram (see Figure 5.1), which consists of three stages: (a) the ontology construction stage (blue) creates a baseline ontology by extracting reusable concepts from multiple privacy policies; (b) the app policy tracing stage (green) yields phrases that describe data types that are collected automatically or from the user directly based on a target app's privacy policy; and (c) the GUI analysis stage (orange) that extracts the input fields, field labels, and view identifiers from the input views, which are then fed to data flow analysis (i.e., FlowDroid [51] is used in GUILeak) as information sources.

The mappings are generated from the ontology, the data type phrases, and the input view IDs and labels, and are used to detect policy violations in the identified input data flows. The mappings allow us to detect two kinds of gaps: (1) when the policy is too abstract (e.g., it refers to "personal information" when the app shares your age and weight), and (2) when data types are collected and shared but they are not described in the policy. In this paper, we are specifically interested in network-targeted input flows in which user provided information flows to any network API method invocation.

We introduce the specific steps in the ontology construction in Section 4.2.1, the input context analysis in Section 4.2.2, and the mapping of policy phrases to GUI labels for violation detection in Section 4.2.3.

4.2.1 Ontology Construction

To construct the ontology, we perform two main steps: (1) construct a privacy policy lexicon from information types extracted from the privacy policies; and (2) identify semantic relationships among phrases in the privacy policy lexicon to yield the ontology. The ontology models three semantic relationships: the *hypernym*, which is an ontological relationship from a more generic

[1] http://repo.xposed.com

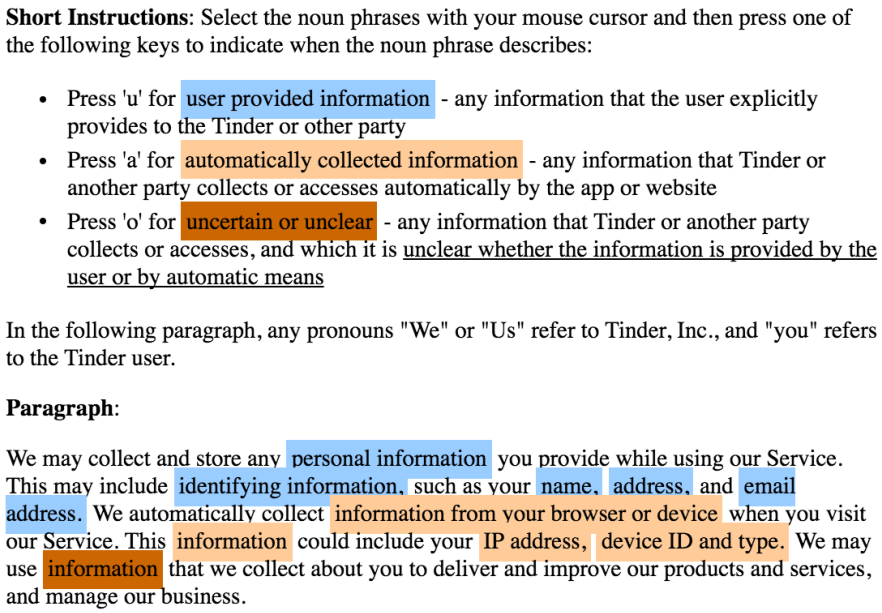


Figure 4.1: Example of Crowd Sourced Policy Annotation Task

concept to a more specific concept; the *meronym*, which is a relationship between a whole and its parts; and the *synonym*, which is a relationship between two concepts with nearly the same meaning. The ontology can be used for automatic violation detection between privacy policies and application code.

Extracting Information Type Phrases

The information type phrase extraction step combines crowdsourcing, content analysis, and natural language processing (NLP) to construct the privacy policy lexicon. For our study, we first select five top applications in each of six sub-categories (personal budget, banks, personal health, insurance-pharmacy, casual and serious dating) in Google Play, to yield 30 total apps for the finance and health categories. Next, we segment the privacy policies into 120 word paragraphs using the method described in [69], which yields annotation tasks from each policy. Figure 4.1 shows an example annotation task, wherein annotators are asked to annotate phrases based on the following coding frame: User-Provided Information; Automatically Collected Information; and Uncertain or Unclear.

The user-provided information annotations describe types explicitly stated in the policies.

However, policies do not always mention how or from whom they collect the information. For example, in Figure 4.1, it is unclear how "information" is collected. To build the privacy policy lexicon, we consider both annotations coded as user-provided information, and uncertain or unclear, in case the policy author described the user-provided collection in an unclear manner. We collect annotations by recruiting five crowd workers from Amazon Mechanical Turk (AMT) to annotate each 120-word paragraph of the combined 30 privacy policies. Because this annotation task differed from [69], we also collected annotations for the same tasks from the authors to evaluate the crowd worker lexicon. Among all annotations collected, we only add annotations to the lexicon where two or more annotators agreed on the annotation. This decision follows the study which shows high precision and recall for two or more annotators [69]. In the next step, we applied an entity extractor [63] to the selected annotations to itemize the information types into unique entities. Finally, the unique information types are added to the finance, health, or dating lexicon depending on which sub-category they belong to.

Identifying Semantic Relationships

Hypernymy is the most common relation found in privacy polices. For example, the concept "personal information" can be used to generalize more specific concepts, such as: "credit card information" or "medications." Therefore, it is important to identify the semantic relationships between the information types found in policies to account for how policies can generally refer to a code-level collection of a more specific information type, as opposed to when those policies omit any mention of the collection practice. To address this issue, we constructed separate ontologies from the finance, health, and dating lexicons. These lexicons share a small number of phrases supporting our decision on constructing individual ontologies for each domain. The ontologies are constructed using the heuristics below [118]:

- Hypernym: $C \sqsubseteq D$, which means concept D is a general category of C, e.g., "password" is a kind of "authentication information."
- Meronym: C Part Of D, which means concept C is a part of concept D, e.g., "email mes-

sage" is a part of “email."

- Modifiers: $C_1\_C_2 \sqsubseteq C_2$ and, $C_1\_C_2 \sqsubseteq C_1\_information$, which means concept C_1 is modifying concept C_2, e.g., “mobile phone number" is a kind of “phone number" and “mobile information."

- Plural: $C \equiv D$, which means concept C is a plural form of concept D, e.g., “addresses" is equivalent to “address."

- Synonym: $C \equiv D$, which means concept C is a synonym of concept D, e.g., “geo-location" is equivalent to “geographic location."

- Thing: $C_1 \equiv C_1\_information$, when concept C has logical boundaries and can be composed of other concepts, e.g., “name" is equivalent to “name information."

- Event: $C_1 \equiv C_1\_information$, when concept C describes an event, e.g., “usage" is equivalent to “usage information."

Semantic relationship identification begins with an ontology, wherein each lexicon phrase is subsumed by the $\top$ concept and no other relationship exists between phrases. Next, two analysts follow four steps (see [118]): (1) they create two copies of the initial ontology $KB1$ and $KB2$, one for each analyst; (2) each analyst individually compares each phrase pair, and creates hypernymy, meronymy, or synonymy axiom between concepts when an appropriate relationship is found based on the above heuristics; (3) the two analysts compare their axioms in $KB1$ and $KB2$ to identify missing axioms and to compute the degree of agreement. Agreement is measured using the chance-corrected inter-rater reliability statistic Fleiss's Kappa; and (4) finally, two analysts meet to investigate the disagreements and reconcile the axioms in $KB1$ and $KB2$. The analysts re-calculate agreement after each reconciliation to measure the improvement due to reconciliation. Identifying semantic relationships is a heuristic-based procedure, wherein each analyst develops their own heuristics or rules for identifying relationships. Once all disagreements are reconciled, the two KBs are equivalent and each one can be used in GUILeak.

4.2.2 Input Context Analysis

The input context analysis stage serves to extract user input views and their contextual UI labels from the app code. In GUILeak, we adapt GATOR [187] to generate a statically-estimated UI view hierarchy, which includes the input identifier, layout, and form elements for each Android activity and dialog. GATOR is a program analysis toolkit for Android that takes the app as its input and produces an estimated XML hierarchy of activities and dialogs. GATOR first generates an event flow graph from code and then iteratively traverses the graph to add views to activities / dialogs (by scanning Android API methods that add views) until a fix point is reached.

For our goal, GATOR has three limitations. First, GATOR does not distinguish between input views and other UI views, so we need to identify and link input views to the API method invocations receiving user input. Second, although GATOR properly collects and inserts the text views holding UI labels in the generated hierarchy, it often cannot provide the UI label values because they are generated at runtime. Thus, we must trace string values back to the string constants defined elsewhere in the code. Third, as shown in Section **??**, common dialogs can be used to receive user input. While GATOR analyzes these dialogs as separate units, they must be linked back to their parent activities so more context information can be extracted. The resulting UI view hierarchies can represent the complete input context. We next introduce how we address these limitations.

Input View Extraction

The first input context analysis step is to extract the API method invocation that receives user input from an input view, such as <android.widget.EditText: android.text.Editable getText()>. These invocations serve as the *sources* in the following information flow analysis. To support this extraction, we carefully reviewed all API methods in subclasses of the Android framework View class, and identified 12 API methods that receive user inputs. The list of these methods is available in our anonymized project website [6]. It is also possible that apps acquire user input implicitly through navigation events, especially when the input is an enumerated type. For example, while static button labels are not user input, a health app may ask a user to click on either a “Male” or

"Female" button, which leads to different subsequent user interfaces. In our research, we focus on the user input views such as text boxes and check boxes, and plan to extend the approach in future to include latent user input.

Next, we insert the user-input-receiving API methods into the view objects contained in the GATOR-generated GUI view hierarchy. This is achieved by inserting code into the view object scanning component of GATOR, so that the user-input-receiving API method invocations are added to the view objects as attributes when a view object is scanned by GATOR. These user-input-receiving API methods are configured into FlowDroid [51] as source API methods. By observing the user data flow within the application from sources to sinks, we can recognize whether the data has been collected from the input-receiving API methods and shared with remote sinks on the Internet. In our analysis, we use the network sinks in SUSI [183].

Input Label Analysis

The second input context analysis step is to extract UI labels in the context of an user input view. In this step, we apply existing string analysis technique [80] to the arguments of all API method invocations that set text to UI views, such as <android.widget.Button: void setText(java.lang.CharSequence)>. Then, we break the value estimation of each argument into a set of strings, so that they can be directly used in the subsequent mapping step. We also link the `setText()` method invocations to GUI views in the view hierarchy using the same approach mentioned in Section 4.2.2

Dialog Insertion

Finally, we insert the dialogs into their parent activities, which allows us to identify the dialog titles and UI labels in the context of each parent. Specifically, we scan the code for dialog-showing method invocations (e.g., <android.app.DialogFragment: void show(...)) and leverage the points-to analysis results from GATOR to discover the dialog types (e.g., GoalSetCheckingInReqDialog). Next, inside the dialog declaration, we collect all the text-setting method invocations in the corresponding builder class and, outside the dialog declaration, we collect all the text-setting methods

invoked on the dialog object. The collected text-setting method invocations are added to the dialog object as attributes. Then GUILeak acquires possible arguments of the text-setting methods with input label analysis, and add them to the view hierarchy of the dialog. Finally, the dialog itself is added as an attribute to the view whose event handlers (identified by GATOR) transitively call the dialog-showing method invocation.

Below, we show a sample dialog insertion result from the extended GATOR; minor details were omitted for space. In the example, we see that the dialog layout was inserted into the parent activity as a sub view of the TextView with title "Set Check-In Requirements."

```
<View ... title="Set Check-In Requirements">
  <View ... title="NO\_TITLE">
    ...
    <View idName="et\_content" ... getValueOp=
    "[<android.widget.EditText:android.text.Editable getText()>]"
     />
    <View ... title="Steps"/>
  </View>
</View>
```

Listing 4.1: Partial Code of Insertion Result

4.2.3 Mapping and Violation Detection

Mobile app privacy policies serve to inform users about which kinds of personal information are collected by apps. Thus, we consider violations as *errors of omission* in that the policy failed to notify the user about a specific, collected information type.

We adopt the definition of *violation* proposed by Slavin et al. [198], which consists of: *weak violation*, which occurs when a policy refers to a vague or abstract information type that semantically includes a more specific type that was omitted from the policy; and *strong violation*, which occurs when the type is completely omitted from the policy. For example, if an app shares a user's medicine intakes, it would be considered a weak violation if the policy only states, "we collect *medical* information. . . ." If the policy neglects to mention medical information as a collected type,

then this omission is classified as a strong violation.

When detecting strong and weak violations, for a input view v whose collected information is sent to network, we first check whether v can be mapped to a concept word c in the ontology. If so, v is considered an input view collecting sensitive information, and we further check whether c and c's ancestors in the ontology appear in the privacy policy. If neither c nor c's ancestors appear, a strong violation is detected, and if any of c's ancestors appear but c does not appear, a weak violation is detected. Slavin et al. [198]'s work uses the similar strategy, but since they focus only on information-accessing API methods, they pre-define a mapping from each API method to a concept work (e.g., mapping the method getLastKnownLocation() to the concept location). Such a predefined mapping is impossible for user-input data, because each app has its own set of input views and they are unknown before analysis of the app. Thus, the core technical challenge we face here is how to map an input view to a concept word in the ontology. Our approach is to develop the conceptual similarity calculation and hierarchical mapping as presented in following two subsections.

Concept Similarity

User-provided information types, including UI labels, are relatively unbounded as compared to platform information types studied by Slavin et. al [198]. Thus, we consider two well-adopted similarity measurements to map UI labels to ontology phrases: WordNet similarity [128] and Cosine similarity [190]. WordNet, which is a popular lexical database used in natural language processing, calculates similarity only for single-word pairs. To accommodate information type phrases that consist of multiple words, we propose a simple greedy alignment as follows: given phrases A and B, we align the word pairs (one in A and the other in B) that have the highest single-word similarity in WordNet, and then perform the alignment recursively until no more words in either A or B remain. For Cosine similarity, we convert the two phrase into two word vectors and apply the standard Cosine similarity formula.

Each mapping between a policy phrase and a UI Label exists, if the similarity between the

phrases, labels and the concept are higher than a given threshold, which is a parameter of our approach. We report results from evaluating this approach under different similarity thresholds in Section 5.3.

Hierarchy Mapping

Unlike API methods, which have explicit meanings, the meaning of user input views are implicit and can be understood only from the context of the view. The input view id can equate to an informative descriptions, but is often inadequate. In our motivating example, the view id of the input box is "et_content", which is too generic to be meaningfully mapped to an ontology concept. Even in three domains (health, finance and dating), and with unbounded user inputs, one still cannot exhaustively iterate all information type phrases for inclusion in the ontology.

To address this issue, we develop a novel mapping strategy that leverages the larger input context, including UI labels, which we call *hierarchical mapping*. In contrast, we refer to the strategy of mapping only the label / id of an input view to concept words as *node mapping*. Our intuition is that, similar to ontologies, the input view hierarchy conveys information about concept relationships between GUI labels. For example, an activity with the title "Transaction Information" may contain multiple user input boxes about transaction time, source account, etc, which all are potential sub-concepts of transaction information in an ontology. Therefore, when mapping user input views to ontology phrases, we use not only the input view ID and label, but also its ancestor view IDs and labels in the view hierarchy.

As shown in Figure 4.2, we first collect the IDs and labels of all ancestor views for a given input view (light blue views). Next, we collect the view IDs and labels that are sibling views immediately before any collected ancestor views (the dark blue view), because sibling views may contain input view labels of their own. We refer to these collected IDs and labels, collectively, as *ancestor labels*. If an input view ID and label cannot be directly mapped to any ontology concept, we further map the ancestor labels to the ontology. Notably, the hierarchical mapping would presumably increase recall due to fewer false negatives, however, at the cost of additional false positives and lower

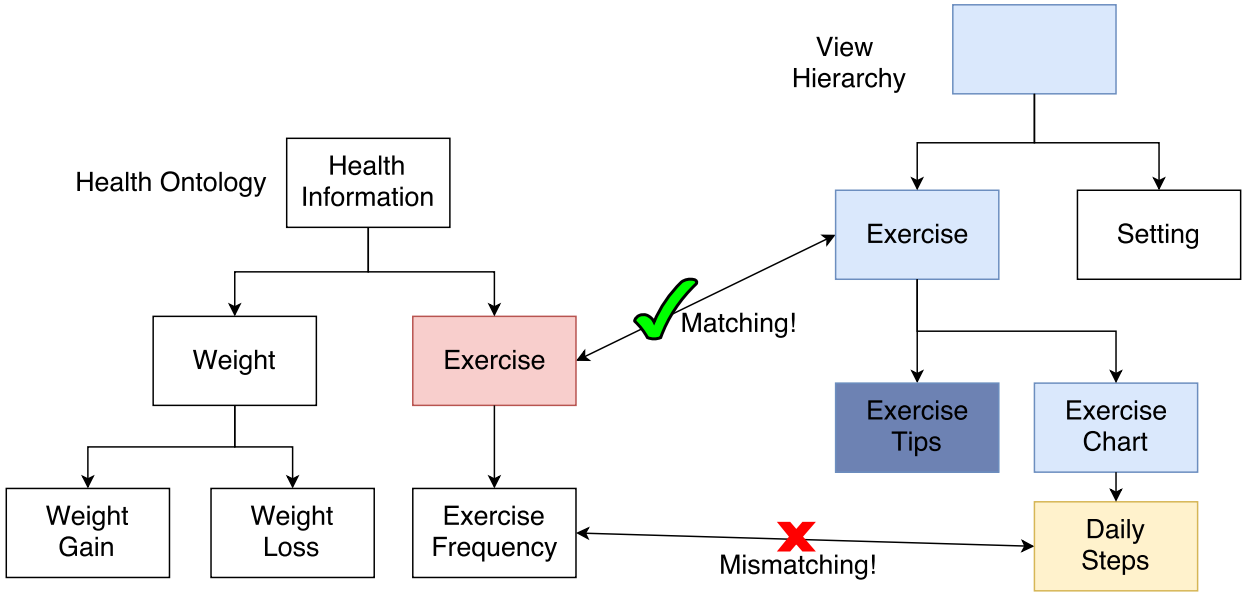


Figure 4.2: Illustration of Hierarchical Mapping

precision. Our evaluation shows that the false positives are less significant when compared with the improvement in recall.

4.3 Evaluation

We evaluate the approach using 150 apps collected from the Google Play with privacy policies in three categories: finance, health, and dating. Within finance and health, we examined apps regulated by GLBA and HIPAA, respectively, and unregulated apps for personal budgeting and personal health. In dating, we explored apps for serious dating, which often requires elaborating user profiles, and casual dating, which includes anonymous chat apps. It should be noted that, health, finance and dating are categories defined in Google Play Market. To be representative, we further classify each category to two sub-categories (personal / institution for finance, fitness / medical for health, and serious / casual for dating). To acquire the 150 apps, in the listed apps in each category at Google Play, we scan from the top until we collect 25 apps with privacy policies for each sub-category. Finance and health apps was collected in Jan 2017, and dating apps were collected in July 2017.

The highest ranked 25 apps that have privacy policies were selected for each category from

Google Play using the category name as the search word. To build our three domain ontologies, we chose the five apps with longest privacy policies from each sub-category (in total 10 apps per category, 30 apps in total). The remaining 120 apps comprise the test set. All data, including the ontologies, links to apps and privacy policies, UI XML files, anonymized survey responses and the violation detection tool can be downloaded at our anonymized project website [6].

Our empirical evaluation tries to answer the following three research questions.

- **RQ1:** What is the effort required and resulting quality from constructing an ontology?
- **RQ2:** What are the quantity and type of privacy-policy violations in apps, if any?
- **RQ3:** How do different similarity calculations and thresholds, and mapping strategies, affect violation detection?

4.3.1 Ontology Evaluation

In response to research question **RQ1**, we report the effort and quality of extracting the lexicon, before reporting effort and quality of constructing the ontology construction. The effort to construct an ontology consists of the time to extract the lexicons from the policies and the time to identify semantic relationships in the lexicons. Table 4.1 shows the total HITs to collect information type annotations, average word count per HIT, total annotations collected from crowd workers and authors, total unique information types extracted, and combined annotation time for authors and crowd workers.

Overall, the average time to extract an information type from a privacy policy in health, finance and dating is 10.6 minutes, 7.0 minutes, and 8.4 minutes, respectively. This time includes the additional time from author annotations needed to evaluate the method.

The lexicon quality is measured by the consensus between author and crowd worker annotations as measured by extracted, unique information types. In health, the authors and crowd workers agreed on 105/198 unique information types. In addition, the authors missed 55 information types that the crowd workers annotated, and the crowd workers missed 34 information types that the

Table 4.1: Lexicon analysis

	Health	Finance	Dating	Overall
Total HITs	141	52	141	334
Average Words per HIT	105	102	116	108
Total Annotations - Crowd Workers	739	309	868	1,916
Total Annotations - Authors	456	198	508	1,162
Total Unique Information Types	197	112	262	490
Annotation Time	34.7	13.1	36	84

authors annotated. In finance, all annotators agreed on 69/112 information types, crowd workers annotated an additional 20 types, whereas authors annotated an additional 23 types. In dating domain, all annotators agreed on 135/262 information types, crowd workers annotated additional 76 information types and authors annotated 51 additional unique information types. Overall, the crowd workers generally identified 18-29% more information types, and authors generally identified 17-20% more types. The consensus was 52-62% of types extracted.

In addition to comparing annotator performance, we compared the lexicon coverage across each domain. The health and finance lexicons share 32/278 phrases, health and dating share 45/415 phrases, and finance and dating share 27/347 phrases. This is an overlap of only 8-12% across three domains, which is due to the differences in policy focus and application features.

Separate ontologies were constructed for each domain where two analysts individually identify semantic relationships between phrases in a lexicon in one domain, followed by a reconciliation step to remove conflicts between annotators. To evaluate the quality of the ontology, we used Fleiss's Kappa to measure the degree of agreement above chance before and after each reconciliation step [69]. The average time per analyst to identify semantic relationships in health, finance and dating was 6 hours, 5 hours and 8 hours, respectively. The average time to reconcile disagreements were 3.7 hours, 2 hours and 5 hours, respectively.

In health, the resulting $KB1$ and $KB2$ for the two analysts contain 951 and 920 axioms, respectively. We obtained these results after two rounds of comparisons and reconciliations. The first comparison produced 491 axioms in disagreement and reconciliation reduced the disagreements to 78 axioms. The Fleiss Kappa after the first and second reconciliations were 0.77 and 0.80, respectively. In finance, the resulting $KB1$ and $KB2$ for the two analysts contain 590 and 582 axioms, respectively. The first comparison produced 292 axioms in disagreement and reconciliation re-

duced the disagreements to 43 axioms. The Fleiss Kappa after the first and second reconciliations were 0.83 and 0.92, respectively, showing a larger increase in agreement. In dating domain, the resulting $KB1$ and $KB2$ for the two analysts contain 1,049 and 1,289 axioms, respectively. The first comparison produced 569 axioms in disagreement and reconciliation reduced the disagreements to 146 axioms. The initial Fleiss Kappa before reconciliation was 0.17 which was increased to 0.79 after the first round of reconciliation.

To evaluate the ontology construction method, two different authors, who we call examiners, independently applied the construction method to the finance lexicon. Before reconciliation, the Fleiss Kappa was 0.14, which is extremely low chance-agreement. After reconciliation, Kappa was increased to 0.83. The Kappa comparing the analyst- and examiner-constructed ontologies was 0.54. On inspection, the disagreement is comprised of 148/474 axioms. Among the 148 axioms, 74 axioms can be inferred using a syntactic analyzer, which automates the process of ontology construction by inferring semantics from lexicon phrases based on their syntax, alone (e.g., plural-singular forms that are equivalent for our purposes). Resolving the 74 axioms with a syntactic analyzer yileds a Kappa of 0.75 between the analysts and examiners. Among the unresolved differences, the examiners found hypernymy relationships missed by the analysts, such as “deposited checks," which are a kind of “transaction.” We believe these differences are due to (1) the various interpretations of phrases by analysts and examiners; (2) the fatigue of comparing phrases; (3) and recency effects that both analysts and examiners experienced during ontology construction [179].

4.3.2 Ground Truth

The research questions **RQ2** and **RQ3** depend on a *ground truth*-the correct number of true violations in the training set. Three challenges must be addressed to establish this ground truth.

First, it is impossible to know all of the true positives, and thus it is impossible to measure recall. To address this challenge, we adopt relative recall [198], which equates the set of true positives to all violations that are detectable with available techniques. In our approach, this violation set results from applying all variants of our approach to the test set, and taking the union of detected

violations. Second, the true mapping from UI labels to the ontology is comprised of the range of input field names acceptable to human interpretation, which can vary. To address this challenge, for each violation detected by any variant of our approach, we first use crowd sourcing to elicit the generally acceptable names of user input fields. Then, we followed rigorous steps to map the crowd worker interpretations of field names to the ontology concepts. Third, the information flows reported by FlowDroid need to be validated with runtime observation of information leaks to the network. We validate the information flows using the Xposed framework. In total, our ground truth construction consists of 21 strong violations and 18 weak violations from 19 apps of the test set.

Eliciting UI Input Field Types

We first analyzed 53 input field labels and found that only 33.9% percent correctly describe the field type. To elicit input field types from crowd workers, we designed a free listing survey [61], in which workers were asked to identify the information type that describes the information entered into the app through a specific UI input field, shown in a red circle in the screenshot (see Figure 4.3). Each survey consists of 3-5 screenshots, and we surveyed 53 input fields from 19 apps. We recruited 30 participants per survey using Amazon Mechanical Turk to yield 393 HITs. Participants of the surveys were located in the United States with an overall HIT approval rating greater than 95%.

We obtained 30 information types per input field. Because there are multiple ways to describe the same concept, we pre-processed the results to more easily compare elicited types as follows: (a) rewrite prepositional phrases into noun phrases, e.g. "amount of money" is rewritten to "money amount;" (b) remove possessives, e.g., "user's current medication" is changed to "user current medication;" (c) replacing "your" with "user", e.g., "setting your own pace" is changed to "setting user own pace;" and (d) remove hyphens, e.g., "e-mail" is changed to "email." These steps are similar to porter stemming in natural language processing, were verb conjugation is removed to make verb comparisons easier [177, 178]. After pre-processing, we combine similar type names for each field and calculate the type name frequency, which is the number of workers who provided

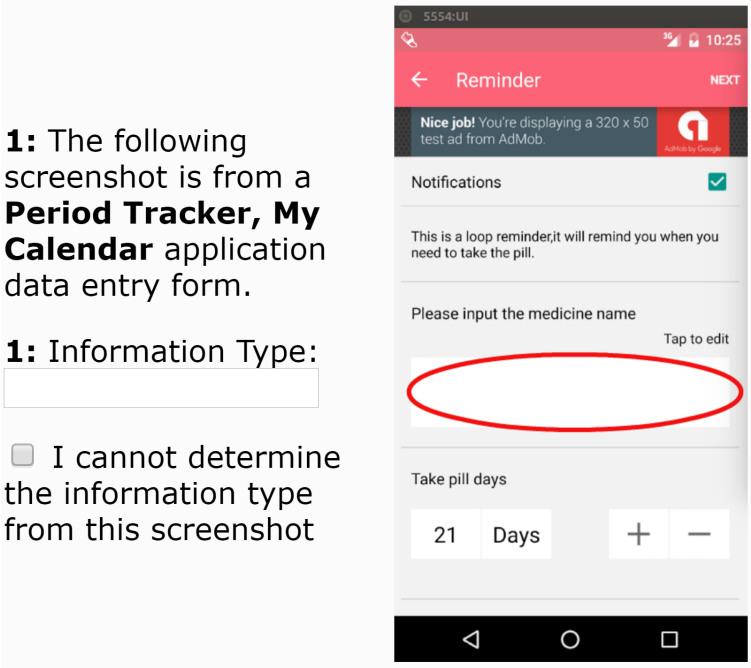


Figure 4.3: User Interface Input Field Tagging Task

each syntactically unique type name per field. Finally, for each field, we select the most frequent type name, which remains linked to a set containing the less frequent type names for that field. This set can be used to expand the interpretation of information types for the same input field, and also can be used to map the UI input field to the ontology, which is described next in Section 4.3.2.

Mapping

We follow a three step approach to map the true input field type names to the ontology concepts: (1) for each elicited name, we look for the exact match of the name in the ontology. If the match is found, we map the name to the matched concept in the ontology; (2) if we cannot find the exact match, (a) we break the name into separate words and create a phrase superset, which includes any combination of the individual words from the name. Next, we look for the exact match of the phrases in the superset with concepts in the ontology. If we find an exact match, we map the original name via this phrase to the matching concept in the ontology; and (b) we identify the purpose for the UI input field name using existing concepts in the ontology, if a matching concept is found for the purpose, we match the name to that concept. (3) If we cannot find a

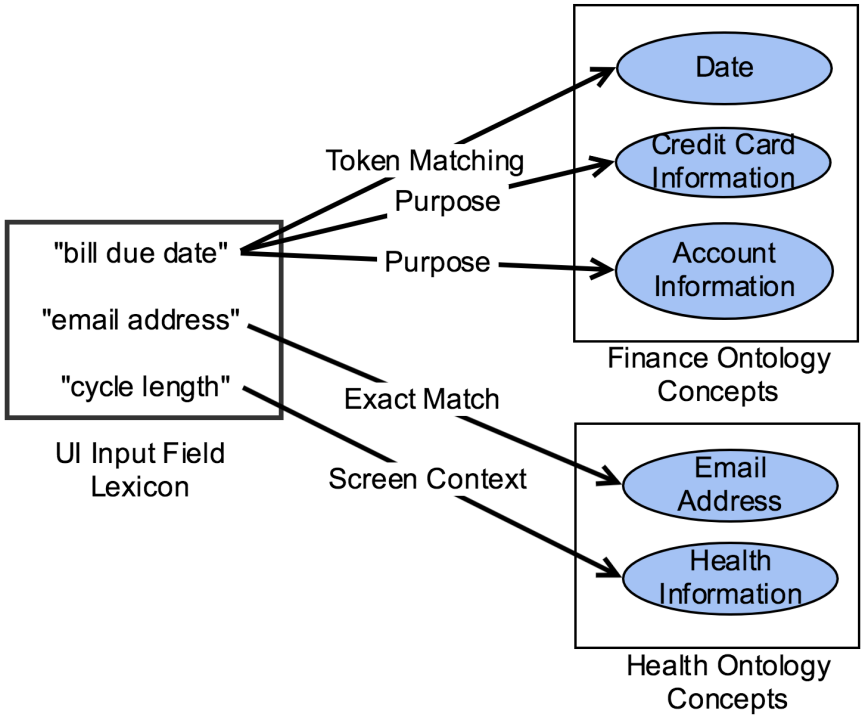


Figure 4.4: Mapping

related concept for the name using steps (1) or (2), we use the context of the screen where the input field is present in the app. The context provides guides to find related ontology concepts to the name. Figure 4.4 shows the mapping for elicited names from the input field in Figure 4.3. The phrase "bill due date" fails to find an exact match in the finance ontology in step (1), but after word tokenization in step (2) produces the word set $S = \{bill, due, date\}$. Next, the superset of S yields $T = \{bill, due, date, billdue, billdate, duedate, billduedate\}$. Finally, the generated name "date" from the superset matches "date" in the finance ontology. In step (3), the purpose for "bill due date" yields matches for "account information" and "credit card information." In a second example, we were unable to find matching concepts for "cycle length" using the two first steps. Therefore, using the context of the screen that contains the UI input field and the application itself, we found that "cycle length" is related to "menstrual information" and not "exercise information." Therefore, we mapped "cycle length" to "health information." This process was performed by the authors who voted on the final result to construct the ground truth.

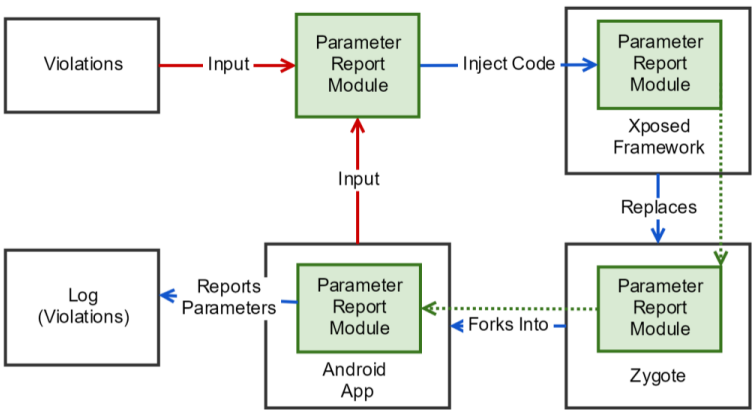


Figure 4.5: Xposed Parameter Reporter Module

Validation with Xposed

We validated the FlowDroid results by implementing a runtime tool to "hijack" the apps with detected violations. To do so, we created a module that utilized the Xposed framework, which is depicted in Figure 4.5. The Xposed framework can modify compiled Android apps at runtime. Xposed takes advantage of the *Zygote* Android daemon, from which all Android apps are forked. By overwriting the process with its own, the framework is able to insert hooks into the bytecode of the app allowing the module to perform custom code before and after hooked method calls. The integration of the module with the app at run time can be seen in the figure starting from the top right box. The module is loaded by the Xposed framework at boot time and thus is transferred into Zygote (which Xposed replaces). When the app is started, the process is forked from Zygote and the module persists within the app's bytecode along with the hooks included for detection of the sinks. Our module inserts custom code before invocations of network sink API methods, and the code simply writes the input parameters for the sinks to a log file. This allows us to trigger the leak of data at runtime and verify that the UI input values were leaked by reviewing log files.

4.3.3 Violation Detection Results

We designed several variants of our approach based on different concept similarity calculations, similarity thresholds, and mapping strategies, which we evaluated using the ground truth. We use *precision*, *relative recall* and *F-score* as our metrics. In our experiment, we consider two

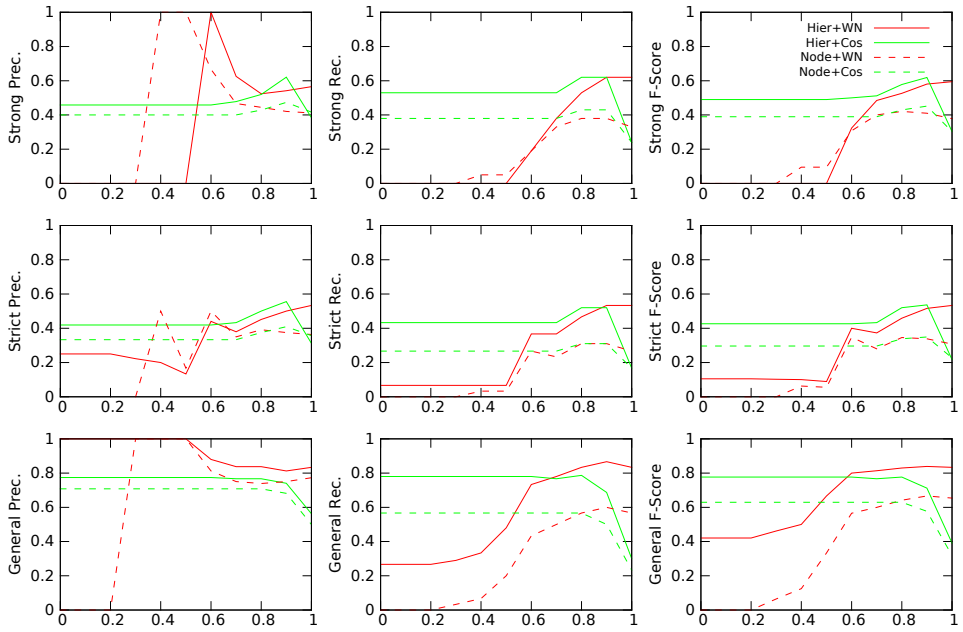


Figure 4.6: Comparison of Technique Variants under Different Similarity Thresholds

similarity measurements: WordNet (WN) and Cosine similarity (Cos). We consider two mapping strategies: node-mapping (Node), wherein only the ID and label of the input view is considered, and hierarchical-mapping (Hier), where IDs and labels of all ancestor input views are considered. This yields four approach variants by combing techniques: Hier+WN, Hier+Cos, Node+WN, Node+Cos.

The violation detection results are presented in Figure 4.6. In each sub-figure, we compare the four variants on different similarity thresholds (0-1), with the legend on the right top corner of the chart. The figures in row 1 compare the precision, relative recall, and F-score on strong violations respectively. The figures in row 2 compare the precision, relative recall, and F-score, respectively, on violations with strict violation types (e.g., strong violations are detected as strong, and weak violations are detected as weak). The figures in row 3 compare the precision, recall, and F-score on violations with general types, respectively. By general types, we mean a violation is considered correctly detected if a true violation is detected but the predicted type is wrong (e.g., strong violations detected as weak, and weak violations detected as strong).

From Figure 4.6, we have the following observations. First, with a 0.8 similarity threshold, our

hierarchical-mapping-based variants can achieve 60% F-score and 65% recall for strong violations, 53% F-score and recall for strict type violation detection, and 84% F-score and 86% recall for general violation detection, where violation-type errors are ignored.

Second, hierarchical-mapping variants (solid lines) perform much better (on average, improved by 20 percent in recall, and 13 percent in F-score) than node-mapping variants in both recall and F-score, while the precision of the two techniques are similar. This observation confirms our intuition that the incorporation of context UI labels greatly improve the violation detection results (note that recall is more important than precision in this scenario, as long as the precision difference is small).

Third, as the similarity threshold increases, the effectiveness of WordNet-based variants improves while the effectiveness of Cosine-based variants reduces. With the similarity threshold around 0.8, all variants are close to their best performance (F-score). One explanation for this observation is that WordNet often provides high similarity scores to words that are not closely related in the domain, but which may be closely related in other domains such as bank and river. Thus, increasing the similarity threshold reduces these false positives. By contrast, Cosine similarity requires matching exact words, and a high similarity threshold will result in losing matches between UI labels and ontology concept names.

Examples. We hereby describe some real examples about privacy violations. To avoid legal issues, we do not reveal the name of apps described. One example of strong violation was found in one of the top pregnancy related health apps with more than 50 million installs. We found that the app sends information about cycles and medicines taken to their servers, but it does not mention sharing this information in the privacy policy. By contrast, a weak violation was found in a top personal budgeting app with more than 1 million installs. We found that this app sends the bill due date information to the server, but it is not directly mentioned in the privacy policy, although "transactions" as a more general concept phrase is mentioned in the policy.

4.3.4 Threats to Validity

On construct validity, the claim that a mobile app violates a privacy policy requires a semantic mapping between the code and a corresponding policy statement. The mapping consists of information type phrases in policies, labels of input fields in mobile app input views, information flows extracted from source code, and formal ontology concepts that align these artifacts. To address this threat, we crowd sourced the identification of relevant policy phrases and for information types of input fields as seen by potential users. In the construction of the lexicon and ontology, we computed inter-rater reliability and compared results across crowd workers and two sets of authors, called analysts and examiners. The method results show a high, above-chance agreement and the method reveals specific sources of disagreement.

On internal validity, the lexicons were constructed from two or more annotators, which yield information types that are acceptable to a subset of annotators, but not all annotators. The liberal interpretation may have skewed our results to include more false positives, in which a privacy policy phrase has a statistically narrower interpretation accepted by the general population than what was accepted in the ontology. In addition, the existing frameworks (e.g., FlowDroid and GATOR) also have deficiencies in precision and recall, which are inherited by our framework. For example, FlowDroid's performance on DroidBench, a benchmark repository of mobile apps, yields 93% recall and 86% precision on data leak detection [51]. To address this threat, we carefully reviewed the flows reported by these frameworks when evaluating the precision and recall of our overall approach.

External validity concerns how well our framework generalizes to other mobile apps and domains. In our study, we chose three different domains aimed at highlighting cross-domain differences: health, finance and dating. In addition to different information types, apps in this domains also present different kinds of input forms and views that were designed to meet different domain-specific features (e.g., diet and exercise versus bank account balances versus social networks).

4.4 Discussion

Bytecode Analysis and Monitoring. Security and privacy research aims to address a specific threat, which in our case is the carelessness of software developers to recognize privacy policy violations in code. Therefore, we assume that these developers have access to the source code, to which they can apply source code analysis and instrumentation techniques. However, in our approach, we choose to perform bytecode analysis and platform-based monitoring, based on the following three reasons. First, our approach allows direct analysis and monitoring of unmodified APK files, which extends access to our framework to a broader community as a third-party service. Thus, developers, project managers, or even regulators can use the framework without access to the source code to check for policy violations. Second, platform-based monitoring is independent from app code changes, which is easier for developers who would not need to re-instrument their code after changes to the app. Third, the existing tools Soot and Xposed reduce the technical difficulty of performing bytecode analysis and platform-based monitoring in comparison to source code analysis and instrumentation.

Limitations on data types. Our approach employs FlowDroid for information flow analysis and SUSI for network sinks. These tools have limitations when detecting flows and network requests involving encrypted data and files. For example, our approach currently cannot detect when the user-provided data is stored in a file and sent out through a network request. With respect to encryption, we consider HTTPS API methods as sinks and, if encryption is performed through HTTPS, our approach can detect the violation. However, if the data is encrypted within the app, FlowDroid may not be able to track the data through data encrypting methods. We believe these limitations can be addressed by improvements so information flow analysis.

4.5 Conclusion

In this work, we proposed a novel approach to detect privacy policy violations due to leak of user input data. To address the two technical challenges (infinite mapping and various GUI

implementation), we adapted the GATOR framework, and developed hierarchical-mapping-based violation detection. We apply our approach on three important domains (finance, health and dating) and detected 21 strong violations and 18 weak violations in 120 popular apps from the domains. Our experiment shows that our best technique variant can achieve a F score of 84% with proper similarity threshold set.

CHAPTER 5: TESTMIG: REUSING OF MULTIPLE GUIS CONSTRAINS

This chapter has published on International Symposium on Software Testing and Analysis (ISSTA) 2019, and was co-authored by Hao Zhong and Xiaoyin Wang.

In this chapter, we will carefully introduce the novel approach we proposed to solve the third challenge problem we mentioned in Chapter 1.

Instead of directly infer the hidden constrains between GUIs, we solve this problem by leverage the exist constrains which is the same app but between different platforms.

Nowadays, Apple iOS and Android are two of the most popular platforms for mobile applications. To attract more users, many software companies and organizations are migrating their applications from one platform to the other, and besides source files, they need to migrate their GUI tests. The migration of GUI tests is tedious and difficult to be automated, since two platforms have subtle differences and there are often few or even no migrated GUI tests for learning. To handle the problem, in this work, we propose a novel approach, TestMig, that migrates GUI tests from iOS to Android, without any migrated code samples. Specifically, TestMig first executes the GUI tests of the iOS version, and records their GUI event sequences. Guided by the iOS GUI events, TestMig explores the Android version of the application to generate the corresponding Android event sequences. We conducted an evaluation on five well known mobile applications: 2048, SimpleNote, Wire, Wikipedia, and WordPress. The results show that averagely TestMig correctly converts 80.2% of recorded iOS UI events to Android UI events and have them successfully executed. In addition, averagely our migrated Android test cases achieve similar statement coverage compared with the original iOS test cases (59.7% vs 60.4%).

5.1 Introduction

Nowadays, a lot of mobile software producers are developing their apps for both Apple iOS and Android. In particular, among the top 10 (July 1^{st}, 2017) third-party apps (i.e., not provided by Google) in the Google Play Store [**?**], 8 apps have their corresponding iOS versions in the

Apple Store. The only two exceptions are Clean Master [25] (a system management tool) and Kika Emoji Keyboard [32] (a keyboard app for inputting emotional symbols), which are closely tied to the underlying system, so that their features cannot be easily migrated. While Android has a market share over 80%, iOS devices and apps are widely reported [23] to have a much higher profit margin. Due to this long-term evenly matched competition between Apple iOS and Android, it is important for software producers to target both platforms for broader user groups.

The iOS and Android versions of an app are typically not developed and released simultaneously. Among the 8 top apps mentioned above with both iOS and Android versions, 5 apps (Snapchat [35], Pandora [34], Instagram [30], Crossy Road [27], and WhatsApp [39]) have their iOS versions released, averagely 9 months before their Android versions are released. The remaining 3 apps (FaceBook [28], FaceBook Messenger [29], and Spotify [36]) have their iOS versions and Android versions released at the same time. FlappyBird [11], one of the most successful mobile application developed by a personal developer, has its iOS version released in May 2013, and its Android version released 7 months later. Some common reasons for the asynchronous development may include the strategical emphasis on users from one platform, the lack of resources or expertise, and limited time.

Due to the above facts, the migration of software cross the two platforms becomes a common and important task in mobile software development, especially from iOS to Android. In literature, many approaches have been proposed to support cross-platform compilation and execution of apps, such as Cordova [24], and Unity3D [38]. However, cross-platform execution is often too inefficient for real-world usage scenarios [24]. Furthermore, while developers can benefit much from code migration tools, the fully automation of behavior-preserving cross-platform code migration is still far from being practical [165]. Therefore, in the practice of code migration across the two platforms, tedious and error-prone manual effort is still unavoidable, and testing is necessary to ensure the quality of migrated apps.

Automatic test generation [47, 149], although solving a more general problem, suffers from various issues (e.g., how to handle logins and generate valid user inputs), and often cannot achieve

sufficient coverage [77]. In this paper, to reduce the testing effort in the specific scenario of application migration, we propose a novel approach, TestMig, to automatically generate GUI tests for an application's Android version, when its iOS GUI tests are available. The two **key insights** behind TestMig are: (1) *to facilitate users, iOS and Android versions of the same application typically have similar GUI structure and interaction patterns (also supported by Joorabchi et al.'s study [125]), so event sequences in an application's iOS GUI tests can be largely reused for its Android version*; and (2) *iOS GUI tests may contain valuable knowledge such as testing accounts for login and meaningful user input data, which helps to resolve well-known limitations in automatic test generation techniques.* The idea of migrating GUI tests is general and applies to both directions of test migration, we implement TestMig to migrate iOS GUI tests to Android GUI tests because (1) facts mentioned above show that iOS versions are often developed earlier, and (2) there are more automatic testing tools (e.g., UIAutomator [37] and MonkeyRunner [33]) for Android, facilitating the implementation of TestMig.

The basic design of TestMig contains three components: the iOS test recorder that records the iOS GUI event sequences triggered by iOS GUI tests, the converter that converts iOS GUI events to Android GUI events, and the explorer that explores the Android version under the guidance of the converter. For each iOS test case to be migrated, the converter and explorer will take the test case's recorded iOS event sequence as input. During the exploration of the Android version, at each GUI state, the explorer sends to the converter a list of Android GUI events that can be triggered at the state, and the converter will tell explorer which event to trigger based on the remaining iOS GUI events in the iOS event sequence. Although conceptually TestMig just translates event sequences from iOS to Android, the test migration process faces the following two major technical challenges.

TC1: GUI design changes. Although application versions on different platforms shall have similar functionalities, their GUIs often have subtle differences, since programmers often change their applications (e.g., replacing tabs in iOS to action bars in Android) to satisfy users' habits. As a result, some mappings between GUI events are not one-to-one (e.g., as shown in Figure 2.3, iOS users need just to tap once on the tab "details", when they fetch the product detail, but Android

users need to tap on the action bar and then tap the item "details" from the drop-down list). In such cases, our converter needs to consider all different compositions of follow-up GUI events to decide the correct event to trigger.

TC2: mapping of GUI controls. To migrate test cases from one platform to the other, mappings between GUI controls in two application versions are necessary but unavailable. Traditional code migration techniques [165, 252, 253] between platforms and languages must collect many cross-platform applications as their training data, when they mine mappings between API methods. Although migrated applications present instances for API mappings, they rarely present the mappings of GUI controls, which are required to migrate GUI tests. As a result, we have to propose an approach that does not rely on migrated code.

To overcome **TC1**, TestMig's converter leverages the sequence transduction technique [161] during the guidance of exploration. Based on a converting probability matrix (called $transduction\ model$) between elements (called $words$) in two domains, the sequence transduction technique synthesizes the optimized sequence in the target domain with overall highest converting probability, while considering many-to-many mapping up to the maximal size of $word$. In our application scenario, we can deem iOS GUI events and Android GUI events as two domains, and the remaining iOS GUI event sequences as element sequences. However, the transduction model still relies on predefined conversion probability between UI events, typically acquired through training which is infeasible as mentioned in **TC2**. To overcome **TC2**, TestMig uses similarity between labels of GUI controls to estimate converting probability between GUI events.

5.2 Approach

In this section, we introduce the design and structure of TestMig, with its overview presented in Figure 5.1. The two inputs of TestMig are the iOS app with GUI tests, and the Android version of the apps. The output of TestMig is migrated Android GUI tests. The migration process consists the following three phases. In the first phase (recording), TestMig records the iOS UI event traces during the execution of iOS GUI tests. In the second phase (exploration), for each recorded iOS

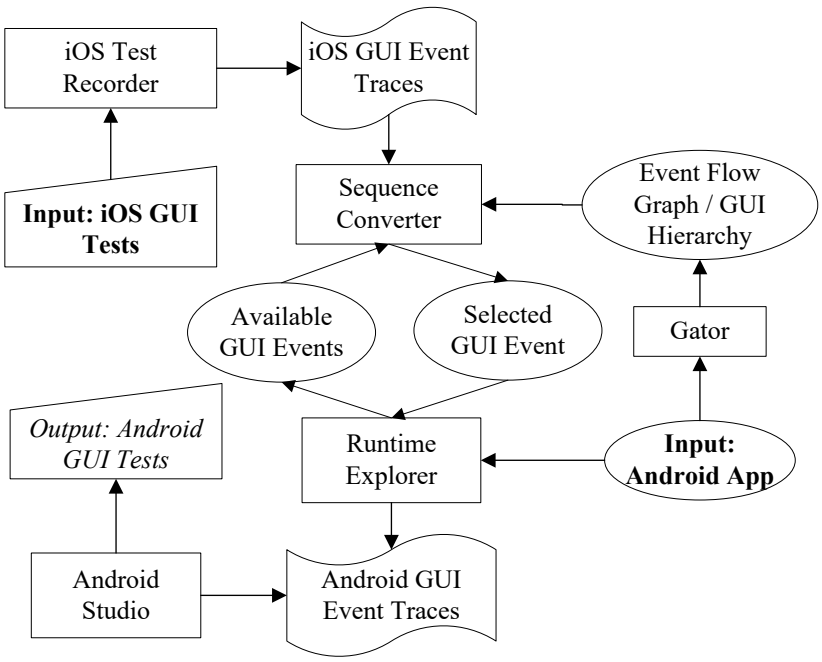


Figure 5.1: Approach Overview

UI event trace tce, TestMig explores the Android version of the app with the guidance of sequence converter which takes tce as its input. In the third phase (generation), TestMig leverages Android Studio to generate Android GUI tests from the Android GUI event trace performed during exploration. The first and third phases are based on existing tools and are thus straightforward, so the core of TestMig is the exploration phase.

At the beginning of the exploration phase, the explorer starts the app. Then at each GUI state, the explorer analyzes the runtime GUI hierarchy to collect a set of available GUI events and send them to the sequence converter. Then, the sequence converter refers to the iOS GUI event trace and a static event flow graph to determine which event should be triggered next. Note that the static event flow graph is generated by GATOR [187] to help TestMig find GUI events available in future for many-to-many event mapping. The sequence converter also needs to determine whether zero, one or multiple events should be consumed from the head of iOS GUI event trace. This process is iteratively performed until (1) the iOS GUI event trace is consumed up, or (2) the exploration cannot continue (i.e., the explorer goes outside the app or no GUI events are available at the current GUI state). The sequence converter is the essential component of TestMig. At each GUI state, its

Algorithm 1 Exploration Algorithm

```
1: procedure CONVERT(I, G, w)
2:     Explorer.startApp()
3:     while I is not Empty do
4:         E ← Explorer.getAvailableEvents()
5:         evt, I' ← SELECTEVENT(I, G, E, w)          ▷
   Select the best event evt, I' is the remaining iOS event
   trace after evt is triggered
6:         if evt is NULL then
7:             break     ▷ Stop Conversion if cannot proceed
8:         end if
9:         success, G' ← Explorer.trigger(evt)
10:        G ← G', I ← I'
11:    end while
12: end procedure
```

Figure 5.2: Algorithm 1: Exploration Algorithm

basic event selection mechanism synthesizes an Android GUI event, which is the most similar one to the prefix of the remaining iOS GUI event sequence. However, as described in the motivation example, the mapping of events is often not one-to-one. To handle the problem, TestMig considers more than one event that appears at the top of the remaining iOS GUi event sequences, and looks further into more than one Android GUI event that can be triggered after the next event. We refer to the number of events to be considered in one mapping as the *transduction length* w.

5.2.1 Runtime Exploration

Our exploration algorithm is presented in Figure 5.2. The three inputs are I, the recorded iOS event trace, G, the static event flow graph generated by GATOR, and w, the transduction length. At the beginning of the conversion, the explorer will start the app (Line 2). Then, while I is not consumed up, the explorer will fetch the set of available GUI events at current GUI state (Line 4), and select a best event to trigger with the $SELECTEVENT$ procedure (Line 5). $SELECTEVENTS$ will also return the remaining iOS GUI event sequence I' by removing the mapped iOS GUI events from the head of I. Then TestMig will trigger evt. Note that when triggering evt, TestMig is able to locate the actual set of available events after evt, so it will refine the estimated event flow graph G to G' based on the information to remove false positives in G (Lines 9 and 10). In addition, it will update I as I' at the end of the iteration (Line 10), and this process

consumes a subsequence of the iOS GUI event sequence. In any case when $SELECTEVENTS$ returns NULL (i.e., when its exploration goes to a dead end or outside the app, the explorer will return an empty set and $SELECTEVENTS$ will return NULL). When it happens, TestMig stops the conversion and reports the migrated sequences (Line 7).

5.2.2 Event Selection

We present the detail process of event selection (i.e., $SELECTEVENTS$ procedure) in Figure 5.3. Our basic idea is to select a pair of iOS GUI event sequence $prefix$ and Android GUI event sequence $path$ (within transduction length w) that has the highest similarity (Lines 10-19). Note that we calculate the similarity based on the transduction probability in sequence transduction, which will be introduced in Section 5.2.3. When calculating the transduction probability (Line 13), we will also retrieve the mappings between events. Then, we return the first event $first$ in the selected sequence $path$, and the updated iOS GUI event sequence I with events mapped to $first$ removed (Lines 16 and 20). Before calculating the similarity, TestMig first collects in P all event sequences in G starting with any event in the available event set E within transduction length w (Lines 5-8). It then collects in H all prefixes of the remaining iOS GUI event sequence I within transduction length w (Line 9). Note that because Gator does not handle fragments now, it may also have some false negatives (some actually trigger-able events are missing from G), and some events in E may not exist in G. To make sure we consider all events in E, we add the full set of E into P when initializing it (Line 5). Finally, note that Figure 5.2 and 5.3 are both conceptual descriptions for clarity. The performance optimization in implementation is not reflected. For example, TestMig stores calculated transduction probabilities so that they are not re-calculated in future event selections.

5.2.3 Transduction Probabilities between GUI Event Sequences

In the event selection procedure, we need to calculate the similarity (i.e., the transduction probability) from an arbitrary iOS GUI event sequence to an arbitrary Android GUI event sequence

Algorithm 2 Event Selection Algorithm

1: **procedure** SELECTEVENTS(I, G, E, w) ▷ I is the remaining iOS GUI event sequence, G and w's meanings are the same as in Algorithm 1, and E is the set of available events
2: **if** E is empty **then**
3: **return** (NULL,NULL)
4: **end if**
5: $P \leftarrow E$
6: **for** Each $e \in E$ **do**
7: $P \leftarrow P \cup$ GETPATHS(G, e, w) ▷ GETPATHS fetches all paths in G starting with e with length up to w
8: **end for**
9: $prefix \leftarrow$ HEAD(I, w) ▷ HEAD fetches I's prefix with length w
10: $max \leftarrow 0$
11: $evt \leftarrow$ NULL
12: **for** Each $path \in P$ **do**
13: $map, prob \leftarrow$ TRANSPROB($path, prefix$)
14: **if** $prob > max$ **then**
15: $first \leftarrow$ HEAD($path$, 1)
16: $evt \leftarrow (first, I - map[first])$
17: $max \leftarrow prob$
18: **end if**
19: **end for**
20: **return** evt

Figure 5.3: Algorithm 2: Event Selection Algorithm

(Line 13 of Figure 5.3). TestMig defines the similarity matrix of the one-to-one mappings between iOS GUI events and Android GUI events, and then calculates the transduction probability between event sequences based on the similarity matrix.

Similarity between GUI Events

Interchangeable UI-Control categories. TestMig uses the similarity between an iOS UI control (C_{iOS}) and an Android UI control (C_{droid}) to denote the probability of converting an event on C_{iOS} to the corresponding event on C_{droid}. For example, the similarity between a button B_{iOS} and a menu item M_{droid} is used to denote the probability of converting a tap event on B_{iOS} to a tap event on M_{droid}. To make sure a specific event appears on both UI-controls, TestMig restricts that the UI-control mapping between UI controls shall accept the same set of UI events. For example, it is unreasonable to map a text box to a button, because the button does not accept input texts.

For the restriction purpose, we define *Interchangeable UI-Control categories* as a collection

of UI controls which accept the same set of operations. For example, a button and a menu item belongs to the same category, because they both accept the tap event (also the long tap event). Specifically, TestMig considers the following four interchangeable UI-control categories: *editables* which include text boxes, date pickers, and number pickers; *clickables* which include buttons, and menu items; *selectables* including check boxes, and drop down lists; and *swipables* which include swipe views, and sliders. We calculate the similarity only for UI controls within an interchangeable UI-control category, so if two UI controls belong to two different categories, their conversion probability is set as 0. It should be noted that, to make our transduction model flexible enough for UI design changes, we make the interchangable UI-control categories rather general.

Similarity Calculation. For each iOS UI control, TestMig calculates its similarity with all the android UI controls in the same interchangeable UI-Control category. In particular, TestMig uses **UI control attributes**, which include the ID, label, and file names of image resources, to generate features for similarity calculation. As most IDs, titles and file names are written in camel names, to make the calculation more robust, TestMig splits all IDs, titles and file names by non-alpha-numeral letters, and at capital letters to generate a list of tokens. Then, its uses these tokens instead of the original value of the IDs, titles and file names as features. To differentiate tokens from different information sources, it adds a header to each token to indicate the token source. For example, if the ID of a UI control is "inputName", the string is split to three features "ID:input" and "ID:Name", in which the header "ID:" indicates that the three tokens are from the ID attribute (not title or file name). After feature generation, TestMig uses the standard TF-IDF formula to weight each feature, and the similarity is calculated with cosine similarity formula.

Empty UI Control. An iOS UI event may not be converted to an Android UI event and vise versa. As described in Figure 2.3, this may happen due to different user habits in iOS and Android. To allow n-to-n mappings (e.g., mapping a sequence of three iOS GUI events to a sequence of two Android GUI events), TestMig introduces a special empty UI control (for both iOS and Android) that can be mapped with any of the four interchangable UI-control categories. However, the empty UI control allows only an empty UI event χ, and the similarity between a UI event and the empty

UI event is defined as a constant p_χ. To minimize the side effect of the empty UI event, we set the similarity from any UI event to the empty UI event as the minimum positive similarity among all UI controls.

Sequence Transduction Probability

After calculating similarities between single GUI events, TestMig calculates the transduction probability between sequences as their similarity, with the probabilistic sequence transduction. Probabilistic sequence transduction [161] is a model that automatically translates an element sequence from one element space to the other. It is widely used in machine translation, speech recognition, bioinformatics and other applications. Consider two spaces of elements $E = \{e_1, e_2, ..., e_M\}$, $F = \{f_1, f_2, ..., f_N\}$, and a sequence S in space E ($s = s_1 s_2 ... s_K, s_i \in E$). A sequence transducer converts S to a sequence in space F by finding the sequence $T = t_1 t_2 ... t_L, T_i \in F$ with the highest conversion probability $p(S|T)$. Specifically, the probability $p(S|T)$ can be recursively calculated with the formulas below.

$$P(\epsilon|\epsilon) = 1 \tag{5.1}$$

$$P(S_{1,K}|T_{1,L}) = (Max_{i=L-W}^{L} P(S_{1,K-1}|T_{1,i}) \times P(s_K|T_{i+1,L}))^{1/Max(K,L)} \tag{5.2}$$

Here, we use $S_{i,j}$ to denote the subsequence of any sequence S from the i^{th} element to the j^{th} element. When $i = j$, $S_{i,j}$ denotes a single element s_i, and when $i = j + 1$, $S_{i,j}$ denotes an empty sequence ϵ. W is the maximal number of elements in F that a single element in E can be converted to (i.e., the transduction length w). The basic idea of the Formula 2 is to split sequence T into two parts in different ways and maximize the probability of converting the first $K - 1$ elements of S to the first part of T and converting the K^{th} element of S to the second part of T. The former probability can be calculated recursively using Formula 2, until the trivial

case in Formula 1 is reached. Finally the value is normalized by the length of sequences mapped (i.e., $Max(K, L)$), so that the formula does not bias to shorter mappings. For $P(s_K|T_{i+1,L})$, we calculate it by inserting empty GUI events χ before and after s_K to map the length of $T_{i+1,L}$. For example, $P(i_1|a_1a_2) = Max(P(i_1|a_1) \times P(\chi|a_2), P(\chi|a_1) \times P(i_1|a_2))$. Since sequence transduction calculates probabilities of all possible mappings among event subsequences, we can also acquire an event mapping with highest probability during sequence transduction, so that we can tell which iOS GUI events the Android event to be triggered is mapped to, and consume those events (Line 16 of Figure 5.3)

5.2.4 Running Example

This subsection describes how TestMig performs sequence conversion, with a running example, i.e., the exploration of WordPress's GUI as shown in Figure 2.3. Figure 5.4 shows a part of a recorded iOS GUI event sequence I (top part) and the corresponding part of the extracted event flow graph G (bottom part). Here we assume the transduction length is 2.

At the beginning of sequence conversion, the remaining iOS GUI event sequence is the whole sequence, and the explorer reaches state 0 of the event flow graph after starting the app. Then the explorer returns three available events `Button:Me`, `Button:Reader`, and `Button:Notifications`. After that, the event selection module crops the event graph G to get the paths within length 2 in G that starts with one of the three events. In particular, we get a_1, a_1a_4, and a_1a_2 starting with event `Button:Reader`. Note that we do not list all the paths in the figure due to space limit. When calculating the transduction probability with length-2 $prefix$ i_1i_2, we can see that the pair $i_1i_2 \rightarrow a_1a_4$ has the highest probability. So a_1 is triggered, and its mapped iOS GUI event i_1 is consumed. TestMig reaches state 1.

Then TestMig tries to convert i_2i_3 from state 1, and the cropped subgraph from G includes a_2, a_2a_3, but not a_4 because a_4 is a false positive of Gator an is automatically removed as TestMig finds it to be not trigger-able at state 1. Note that if Gator does not report a_4 as false positive, TestMig still gets state 1 because the pair $i_1i_2 \rightarrow a_1\chi$ has the second highest transduction at state

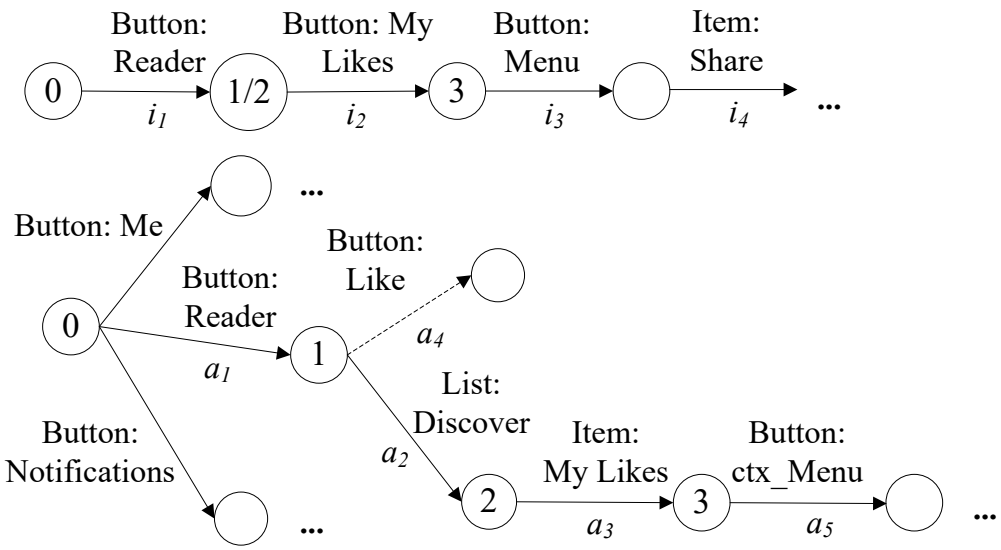


Figure 5.4: Selection of the next GUI Action

0. Recall that the empty event χ can be mapped with any event during sequence transduction. During sequence transduction, although i_2 cannot be mapped to a_2, but the sequence $\chi i_2 i_3$ can be mapped to $a_2 a_3 \chi$, with i_2 mapped to a_3. So a_2 is triggered but no iOS GUI event is consumed as a_2 is mapped to χ, and TestMig reaches state 2. From state 2, pair $i_2 i_3 \rightarrow a_3 a_5$ has the highest transduction probability so a_3 is triggered and i_2 is consumed.

To show the power of TestMig on selecting the correct path, let us assume that a_4 is not a false positive but a trigger-able event. In such a case, at state 1, a_4 has a higher similarity with i_2. However, since we perform sequence transduction, the probability of pair $\chi i_2 i_3 \rightarrow a_2 a_3 \chi$ is $P_\chi^{2/3} \approx 0.13$. This value is comparable with the probability of pair $i_2 i_3 \rightarrow a_4 \chi$, which is $(P_\chi \times P(\text{“Posts I Like", “Like"}))^{1/2} \approx 0.17$, although we will still fail to choose the correct path. However, when the transduction length becomes 3, the probability of pair $\chi i_2 i_3 i_4 \rightarrow a_2 a_3 a_4 \chi$ will be $(P_\chi^2 \times P(\text{“ctx_Menu", “Menu"}))^{1/4} \approx 0.20$. This is higher than the probability of pair $i_2 i_3 i_4 \rightarrow a_4 \chi \chi$, which is $(P_\chi^2 \times P(\text{“Posts I Like", “Like"}))^{1/3} \approx 0.11$. So the correct path $a_2 a_3 a_4$ is chosen.

Table 5.1: Evaluation Subjects

Project Name	Domain	Size (LOC)	#Test Cases	#GUI Events	iOS	Android
2048	Game	1.9K	5	59	30a0f15	0fd6786
SimpleNote	Notepad	17.9K	6	130	4.4.2	1.5.7
Wikipeia	Knowledge media	59.5K	42	475	5.6.0	2.6.203
Wire	Communication	95.7K	32	316	2.41	2.41.359
WordPress	Personal blog	115.3K	31	431	8.3	8.3

5.3 Empirical Evaluation

To evaluate our approach, we conducted an empirical evaluation on five open source mobile software projects. Specifically, we try to answer the following three research questions:

- **RQ1:** How effectiveness is our approach on migrating UI test cases?
- **RQ2:** How does the parameter of our approach, i.e., the transduction length (w), influence the effectiveness of our approach?
- **RQ3:** Why does our approach fail to migrate some UI test cases?

5.3.1 Study Setup

In our evaluation, we used five popular open source applications which have both iOS and Android versions[1]. Among the five applications, 2048, SimpleNote, and Wikipedia have iOS GUI tests in the code base. WordPress has a small GUI test suite with only 5 test cases. Wire has a GUI test suite but it is not in its code base and the developers do not want to make it open. Therefore, we manually enriched the test suite of WordPress, and prepared a test suite for Wire. Please note that it is not easy to find usable experimental subjects because many applications are not open source. Even if an application is open source, it can use libraries that are not available. Although we cannot select them as subjects, close-source apps can use TestMig in their development, since their source files are available in their own programming contexts.

[1] All the source code and GUI tests we used in our evaluation are packaged and available at the anonymized project website https://sites.google.com/site/testmigicse2019

The basic information of the projects are presented in Table 5.1. In Columns 1-5 of the table, we present the subject's name, domain, size (Lines Of Code), number of GUI test cases, and number of GUI events triggered in iOS GUI test execution, respectively. From the table, we found that these subjects cover five different major app categories, with their iOS versions' sizes ranging from 1.9k to 95.7K lines of code. For WordPress, a portion of the app's GUI are written with web views, and are thus not covered by the original iOS GUI tests. Therefore, when calculating coverage, we excluded the code portion only reachable from web views (based on a conservative call-graph), and considered only 80.8K lines of code. In Columns 6 and 7, we present the iOS version (commit ID) and Android version (commit iD) we used in our evaluation. When choosing Android versions, we always use the stable Android version (if there exists one) or commit ID after the iOS version within smallest time gap.

During our evaluation, for each subject, we compiled the iOS version in XCode with iOS 10.3 and Swift 3, and executed the test cases on iPhone 7 simulator to acquire logs containing GUI event sequences. Then, we used TestMig to explore the corresponding Android app according to each GUI event sequence (each GUI event sequence corresponds to a test case). For each generated Android test case, we executed it and manually examined whether it performed the same GUI interaction as the iOS test case from which it was migrated. A test case is considered successfully migrated only if all UI events in the iOS test case are correctly mapped to android UI events (including correctly mapping an iOS UI event to an empty UI event), and all mapped android UI events are successfully invoked. In the manual examination, we have two students as evaluators to watch both executions of iOS test case and the migrated Android test case, and report at which GUI event the two test cases start to differ (including the case where one test case terminates). When the two evaluators report different events, the second author will watch executions again and make the final decision.

5.3.2 Measurements

To answer question RQ1, we need to develop a number of measurements to measure the effectiveness of the test case migration. The most straightforward measurement is the *Test Migration Rate (TMR)*, which measures what proportion of iOS test cases are successfully migrated.

TMR considers only fully successfully migrated tests. In reality, if a test case is partially migrated, it may still cover some code and find some bugs. We consider an iOS test case to be partially migrated if the first k (k larger than 0) iOS UI events are correctly mapped to Android UI events and are correctly executed. For a given iOS test t as a GUI event sequence, we use $partial(t)$ to denote the length of longest prefix of t that are correctly mapped and executed, and $length(t)$ to denote the length of t. We then calculate the migrated proportion of test t ($prop(t)$) as $partial(t)/length(t)$, and define the *Partial Migration Rate (PMR)* of a set of test cases as the arithmetic average of $prop(t)$ for all test case t in the test suite. Finally, we define PMR of a test suite $T = t_1, t_2, ..., t_n$ in formula below:

$$PMR = \frac{\sum_{i=1}^{n} prop(t)}{n} \tag{5.3}$$

From the formula, we found that PMR deems all test cases with equal weights. As test cases with longer GUI event sequences may provide more code coverage, a migration successful rate on GUI events can be helpful. We further defined *Event Migration Rate (EMR)* of a test suite as follow:

$$EMR = \frac{\sum_{i=1}^{n} partial(t_i)}{\sum_{i=1}^{n} length(t_i)} \tag{5.4}$$

Since the ultimate goal of the test case migration is to cover the code of the Android version, we use the test coverage of the generated Android test cases as a side measurement for the effectiveness of our approach. In particular, we use statement coverage (reported by Jacoco [31]) as XCode also reports statement coverage of the original iOS tests so we are able to compare the coverage values.

Table 5.2: Migrated Android test cases

Project	Size (LOC)	#PMT	#Event
2048	1.7K	5	54
SimpleNote	9.3K	6	88
Wikipeia	73.2K	42	433
Wire	71.2K	32	240
WordPress	109.2K (77.5K)	31	404

5.3.3 Migration Effectiveness

The basic information of the migrated Android test cases is presented in Table 5.2. Columns 1-4 provides the subject's name, size of Android version, number of partially migrated tests, and number of GUI events invoked in the migrated tests, respectively. Comparing the table with Table 5.1, we have the following observations. First, the Android versions of 2048, Wikipedia, and WordPress have similar sizes with their iOS versions, but the Android versions of SimpleNote and Wire are much smaller than their iOS versions. This reflects that some features may be missing in their Android versions, compared with their iOS versions. This may have impact in the test migration results as we will introduce later. Second, TestMig is able to fully or partially migrate all iOS GUI tests, which shows that the iOS and the Android versions of the five subject apps all have very similar main activities. Third, the migrated tests trigger less events than original tests as some tests are not fully migrated. Note that the numbers in Column 4 cannot be directly used for calculating EMR, as one iOS events can be translated to 0 to W (longest transduction length) Android GUI events.

Our evaluation results on migration rates are presented in Figure 5.5. In the figure, for each subject, the three columns from left to right represent the TMR, PMR, and EMR of the project, respectively.

From the figure, we found that, in 2048, Wikipedia, and WordPress, TestMig achieves over 80% on all three migration rates: TMR values are 80.0%, 83.3%, and 83.9%; PMR values are 88.0%, 91.2%, and 89.1%; and EMR values are 89.8%, 89.3%, and 87.5%. Generally, if an EMR value of a project is more than its PMR value, it denotes that TestMig achieves better results on

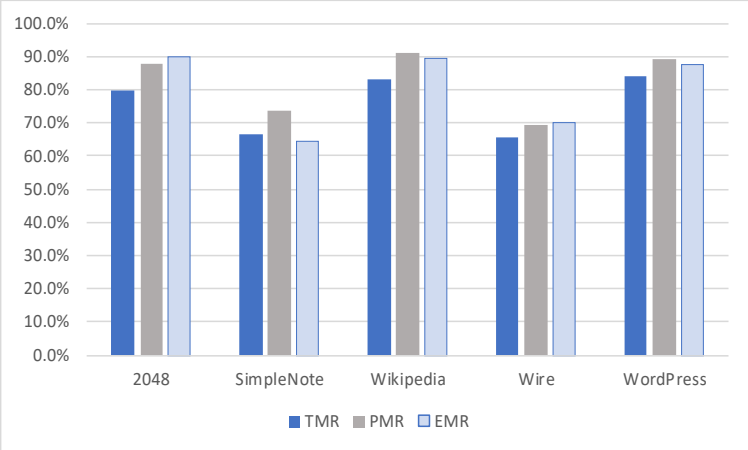


Figure 5.5: The overall TMR, PMR, and EMR

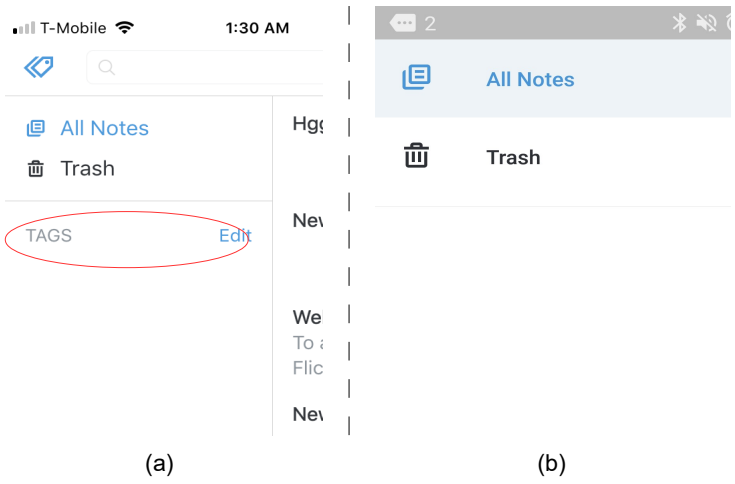


Figure 5.6: Example of a missing feature

longer test cases, and vice versa. For SimpleNote and Wire, our approach has lower migration rates with 66.7% and 65.6% TMR, 73.5% and 69.6% PMR, and 64.6% and 69.9% EMR. We found that the the Android versions of SimpleNote and Wire are still under development, and some modules are not migrated from their iOS versions. As TestMig cannot migrate a test case that invokes a non-existing modules, it achieves relatively low PMR and EMR values. However, this actually does not hurt the effectiveness of migrated tests, and non-existing moduless do not need to be tested. In Figure 5.6, we present an exemplar missing modules in SimpleNote, where the "Tag Edit" button appears in the iOS version (the left screenshot), but does not appear in the Android version (the right screenshot).

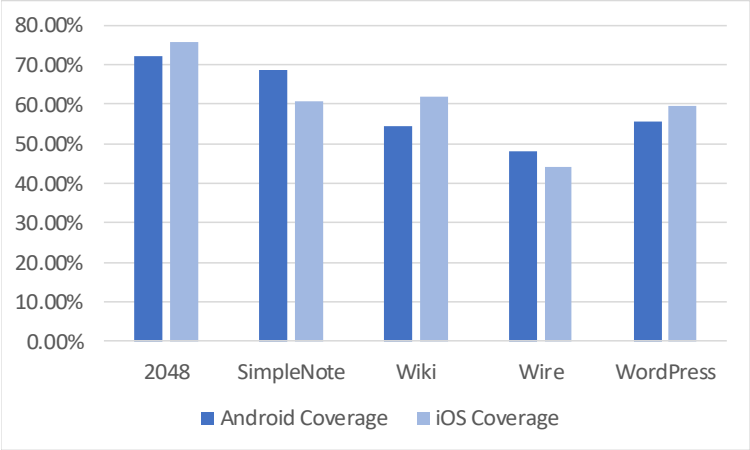


Figure 5.7: Statement Coverage

The test coverage of the migrated Android tests are presented in Figure 5.7. In the figure, for each subject, the left column represents the test coverage of the iOS test cases on the iOS version, and the right column represents the test coverage of the migrated test cases on the Android version.

From the figure, we found that, for the three apps such as 2048, Wikipedia, and WordPress, our migrated Android tests achieved almost the same coverage as their iOS tests. In particular, the migrated tests achieve test coverage of 72.3%, 54.2%, and 55.4%, compared with the test coverage of 75.6%, 62.0%, and 59.6% for the original iOS tests. An interest finding is that in SimpleNote and Wire, which have lower migration rates, our migrated test cases achieve even higher coverage than the original iOS tests. As we mentioned earlier, the Android versions of SimpleNote and Wire are still under development, so it is easier to achieve a higher code coverage as their total code sizes are smaller. By contrast, Wikipedia has a larger Android version than iOS version, so the achieved test coverage is relatively low despite the high migration rates. However, we believe that the cases of Wire and SimpleNote are more common for real-world migration scenarios, as the migrated versions tends to have fewer features than the original versions, at least during or shortly after the migration, when test migration is mostly required.

Here, the test coverage of the original iOS GUI tests is not very high (ranging from 44.3% to 75.6%), so the test coverage of the generated Android test cases are also not very high. However, as revealed in previous studies [223], manual tests often achieve relatively lower coverage but cover

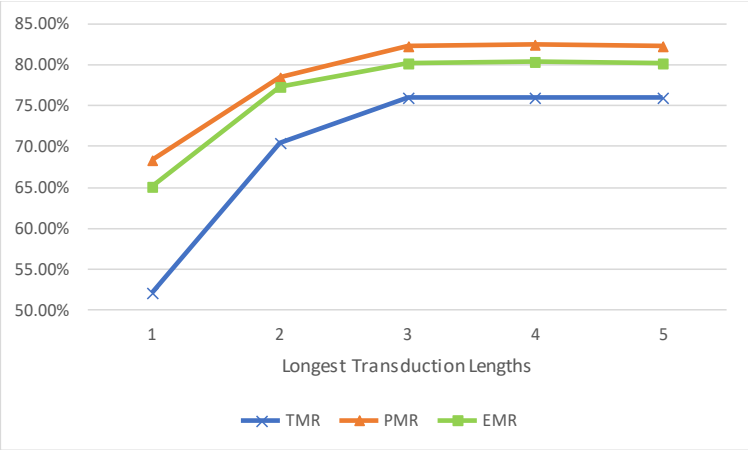


Figure 5.8: The impact of W on migration rates

most popular and error-prone software features. Our approach achieves similar test coverage (not only coverage values, but also the covered code) on the new platform.

5.3.4 Impact of Transduction Lengths

To explore RQ2, we studied how the average TMR, PMR, and EMR of four subjects change, when the transduction length w increases from 1 to 5. Here, we consider w value from only 1 to 5, since it is unlikely that an iOS GUI event is translated more than five Android GUI events, according to our inspection.

Figure 5.8 shows the results. From the figure, we can observe that, as w increases, all three migration rates grows more and more slowly, and the migration rate becomes very stable after w reaches 3. This actually shows that almost all successfully translated iOS GUI events are mapped to at most three Android events. Therefore, we believe that 3 is a proper value for longest transduction sequence, and in our evaluation we use w=3 as the default value of TestMig. Furthermore, the figure shows that TestMig gains a lot on all migration rates (18.2, 10.2, and 12.3 percentage points gain for TMR, PMR, and EMR, respectively) when w increases from 1 to 2. The result shows that one-to-one event match is not sufficient, and our sequence-transduction-based exploration mechanism is helpful.

5.3.5 Categorization of Migration Failures

In our evaluation, we failed to fully migrate 26 out of 116 test cases. The main reason is that the generated Android UI event sequence stuck at a certain place and cannot invoke the next UI event, or the event sequence goes to a "wrong way". It should be noted that, although failures in code migration may significantly undermine the technique's usability, failures in test migration may not affect usability much, as long as the failing rate is relatively low. The reason is that, an unsuccessfully migrated test case is still a test case and may detect bugs in the target app version, although it may not be as good as the successfully migrated ones, and miss part of the consistency checking between versions. Since it partially leverages information from the original test, it may be still better than automatically generated test cases (e.g., on passing log-in guard). Furthermore, as shown in Figure 5.5, our PMR values are larger than TMR values, indicating that TestMig may have successfully migrated a large portion of a test case before the failure happens.

To answer question RQ3, we further investigated the these, and found the root cause mainly falls into 3 categories.

Missing Features. The first category of migration failures are due to missing features in Android versions, so that an original GUI event cannot be matched to anything. This reason accounts for 14 out of 26 migration failures, including 8 migration failures in SimpleNote and Wire. Please note that these migration failures are harmless, because the testing and migration themselves are unnecessary due to the missing features.

System Events. The second category of failures are due to TestMig's inability to handle system events, so the Android tests will stuck when a system event is simulated in iOS tests. This accounts for 8 out of 26 migration failures, including 5 migration failures from Wire, which is a communication software and relies more on system events. In the future, we plan to further map system events between Android and iOS to reduce such failures.

Similar UI Events. The third category of failures happen when multiple GUI events will enable similar following GUI events (accounting for 3 migration failures). Longer transduction lengths will typically solve such issues, but when the future events are not visible by GATOR,

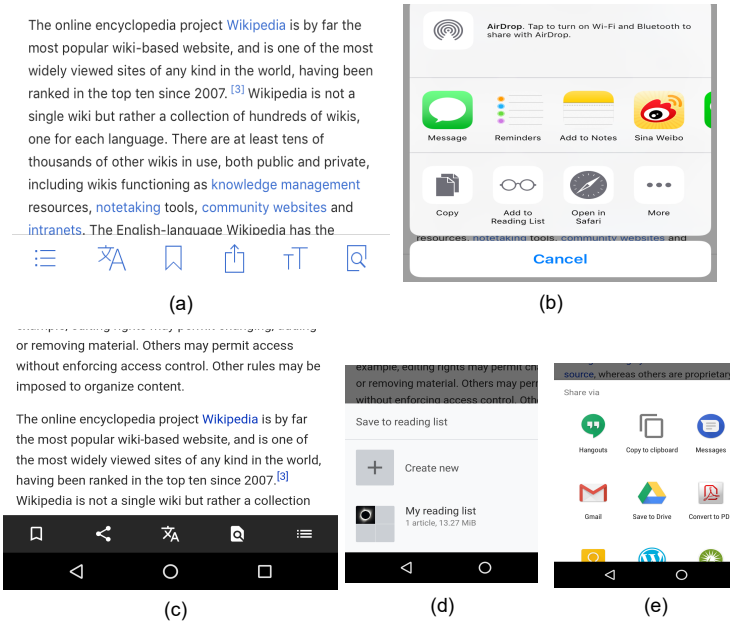


Figure 5.9: Example of a misleading GUI event

TestMig will make mistake. Figure 5.9 shows an example from Wikipedia, where the user is reading an article. Figure 5.9a and Figures 5.9b are screenshots from iOS, and the rest are from Android. In this example, the menu at the bottom of Figure 5.9a and Figures 5.9b have different orders in iOS and Android. In the iOS version, the third and fourth icons from left, triggers "save to history" and "share", respectively. In the iOS version, they are the first and second icons. All the icons have simple ids as `item1` through `item5` / `item6`, so the similarity measurement does not work on them, and we have to rely on the sequence transduction to find the similarity on following GUI events. In the Android version, the following event of clicking on the "save to history" button is "save to reading list", which is presented in Figures 5.9d. However, in the iOS version, this feature happen to be deleted, and the "share" feature is followed by the screen shot in Figures 5.9b, which can trigger a GUI event to "Add to reading list". Here, as the reading list in the iOS version is the reading list of the system instead of the Wiki app, it is a part of "share" feature. In such a case, TestMig mistakenly explores to the "save to list" feature, instead of the "share" feature in the original iOS test case. Since the events after "share" are outside the app, they are not visible to GATOR, and TestMig cannot get sufficient guidance. To reduce such failure, it is possible to

extend GATOR with some common system GUI events for further guidance of TestMig.

5.3.6 Threats to Validity

There are two major threats to the internal validity of our evaluation. First, there can be mistakes in our data processing and bugs in the implementation of TestMig. To reduce this threat, we carefully double checked all the code and data in our evaluation. Second, the manual validation of test and event migrations may involve subjective bias. From our experience, the GUI semantics of the iOS version and the Android version are very similar although the design can be different. As a result, it is not difficult to determine whether two features from two versions can be matched to each other. We further reduce this threat by having multiple evaluators examining the test executions. The major threat to the external validity of our evaluation is that our conclusion may apply to only the data set being evaluated. To reduce this threat, we use four different popular open source projects to cover more testing scenarios.

5.4 Discussion

Migration of UI Test Oracles. Automatic test oracle is an essential part in automatic software testing, but has been one of the most difficult problem in the area. In our approach, we focus on the migration of iOS UI event traces to explore the Android app, and do not consider the migration of test oracles (e.g., assertions in the iOS test script). However, just as using any other automatic UI-test-case generation tool, developers can still use with our approach the general automatic UI test oracles such as crashes, unhandled exceptions, bad presentation, and unresponsive UI controls.

We observe at least two more technical challenges in the migration of UI test oracles. First of all, test-oracle migration requires more precise mapping of UI controls cross platforms. In the transduction of UI event traces, we can explore the Android app with multiple mapping options to double confirm the mapping. This is not possible for test oracles and the imprecision in mapping will cause imprecision in the test results. Second, values in the UI test assertions may be platform specific. For example, some assertions refer to absolute positions of UI controls and padding sizes,

which are affected by the resolutions and various default settings which are different between iOS and Android. However, despite the technical challenges, migration of UI test oracles is still potentially feasible. We plan to work in near future on overcoming the challenges mentioned above and examining the feasibility of UI-test-oracle migration.

UI-Test-Case Migration in General. Migration of UI test cases among various UI frameworks is a general problem, and our approach just solves an specific instance of migrating iOS UI test cases to Android. A panacea for all UI-test-case migration would be having a unified computation platform for all software applications. However, the history of software engineering has witnessed so much competition among platforms, frameworks, and design models. Novel techniques such as web browsers, smart phones, and virtual reality devices also bring in new UI interaction requirements and features. Therefore, we believe that the requirement for UI-test-case migration may exist for a long time. The high level ideas of our approach, including the mapping of UI controls, and transduction of UI event sequences, is applicable to most UI-test-case migration scenarios. As mentioned in Chapter 2, due to the small number and commonality of UI control types in different frameworks, the extension of our approach to more frameworks may not cost much manual effort. However, migration tests beyond mobile platform may bring in more challenges. For example, converting desktop UI test cases to mobile UI test cases needs more consideration in UI design changes, such as the lack of "swipe" operation in desktop UI, and the lack of "drag" operation in mobile UI. When migrating web UI test cases, the extraction of UI controls and their attributes from HTML and Javascript can also be a challenging problem.

5.5 Conclusion

In this work, we propose a novel approach that migrates iOS UI test cases to Android UI test cases. Specifically, we record the iOS UI event sequences invoked during the iOS UI testing, and use the sequence transduction technique to convert this sequence to a sequence of Android UI events. Then, we explore the Android version of the software with the Android UI events and record the test cases. We evaluate our approach on four popular cross platform mobile apps, and

the result shows that our approach can successfully convert averagely 80.2% of the iOS UI test cases to Android test cases, and achieve an average test coverage of 59.7%, which is close to the 60.4% average test coverage of the original iOS test cases on the iOS versions of the apps.

CHAPTER 6: FUTURE DIRECTIONS

In this chapter, we will briefly discuss the future directions of the UI analysis and exploration. And I will shortly report the preliminary work I have done.

As we discussed in Chapter 2, there are three types of GUI Testing and most research works are focused on script testing and exploratory testing. The ultimate goal for GUI testing is to develop an automated GUI testing approach that meets all the testing criterion and pass the testing evaluations. One major obstacle for achieving this goal right now is lacking of meaningful test case generation. Without the meaningful test cases, test cannot run by purpose, thus the test results can no longer be evaluated.

Our future direction is to apply the UI analysis techniques and generate meaningful test case that can improve the current state-of-art. Generating the meaningful test cases including following directions:

- **Generate meaningful user input.** GUIs usually depend on user's input and respond the customized services. Meaningful user input will result in the meaningful response and thus we will cover more functionalities of the application under testing.

- **Generate meaningful UI events and Oracles.** Meaningful UI events means reasonable interactions. Application is usually designed to serve human beings. A more human like interaction will help lift the manual testing burden. And meaningful oracles usually come after meaningful interactions. For each UI event we generated, there should be an expected reasonable response which can be evaluated.

In the following sections, I will introduce the preliminary work I have done on generating meaningful user input.

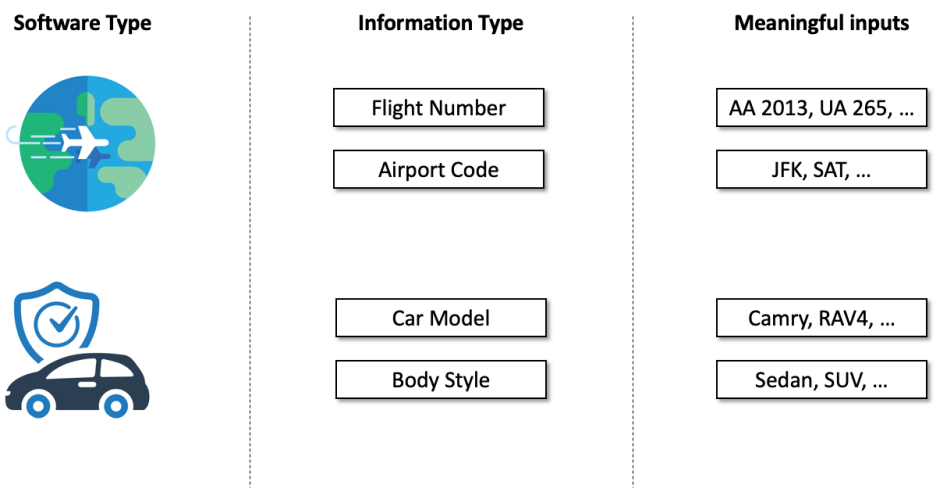


Figure 6.1: Generating Meaningful Input Example

6.1 Generate Meaningful User Input

As discussed in Chapter 1, An UI element with text labels usually come with an information type. This information type can be directly inferred from text label alone and most of the time, it needs to be understood by context. And when this UI element requires an input, for example, an input box, then generating a meaningful user input becomes generating a meaningful information instance for this information type. Through the UI label analysis, we already have the context information, and the next thing we need to do is to infer an reasonable information type [66] and find an instance for it.

Figure 6.1 shows examples of the meaningful input. For example, if an user is using the travel app to search a flight, the app usually requires him/her to enter the flight number or airport code. The meaningful input instances for these two types will be somethings like UA 265 and JFK respectively. And if you only provide a random string with numbers, which is the current solution for automation testing, what is the percentage of possibility to get an reasonable search result? Similar situations happen in auto apps. If you try to search and purchase a car, you have to enter the meaningful car model and body style.

In general, since we have so many types of input, generating meaning inputs including generating meaningful text input, meaningful voice input, meaningful gestures and movement input for

VR, and many more with new IoT techniques. We narrow down our current research direction by focusing on generating meaningful text input in Android application, and it can be extended in the future.

6.2 Generate Meaningful Text Input in Android Application

We want to develop a new framework to generate the text input but base on the existing state-of-art in GUI testing approach. In this way, we could leverage the best testing coverage to maximum the usage of our approach. Since script testing is mainly written by human testers, we will use the exploratory testing as our baseline. The basic idea is that we are going to intercept the testing exploration and provide the meaningful input, while keeping the performance of the exploration.

Monkey [1] is so far the best tool for random exploration based on existing studies [77] [220]. This is because of the two natures of the Monkey: **Robustness** and **Efficiency**. Robustness is because Monkey can handle complicated and unprecedented GUI structures, such as embedded web views, animation, and game objects. while the current research tools can only work on limited versions or types of the GUI framework. Efficiency is that Monkey produces GUI interactions in an tremendous high volume than any other tools and it mostly alleviates its inefficient exploration and brings in an higher coverage as a result.

We proposed a new framework named MONKEY++, a random exploration tool with interaction-pattern guidance, currently focusing on text input filling from input box. The overview structure is shown in Figure 6.2. Monkey is running in Explorer model, while two controllers, Sampler and Activity Switch Detector handling the stop signal for Monkey. As long as there has an input box been detected, the MONKEY++ will stop Monkey and filling the input box with text. So far we are providing the default text input, and we will work on providing meaningful textual input in the future.

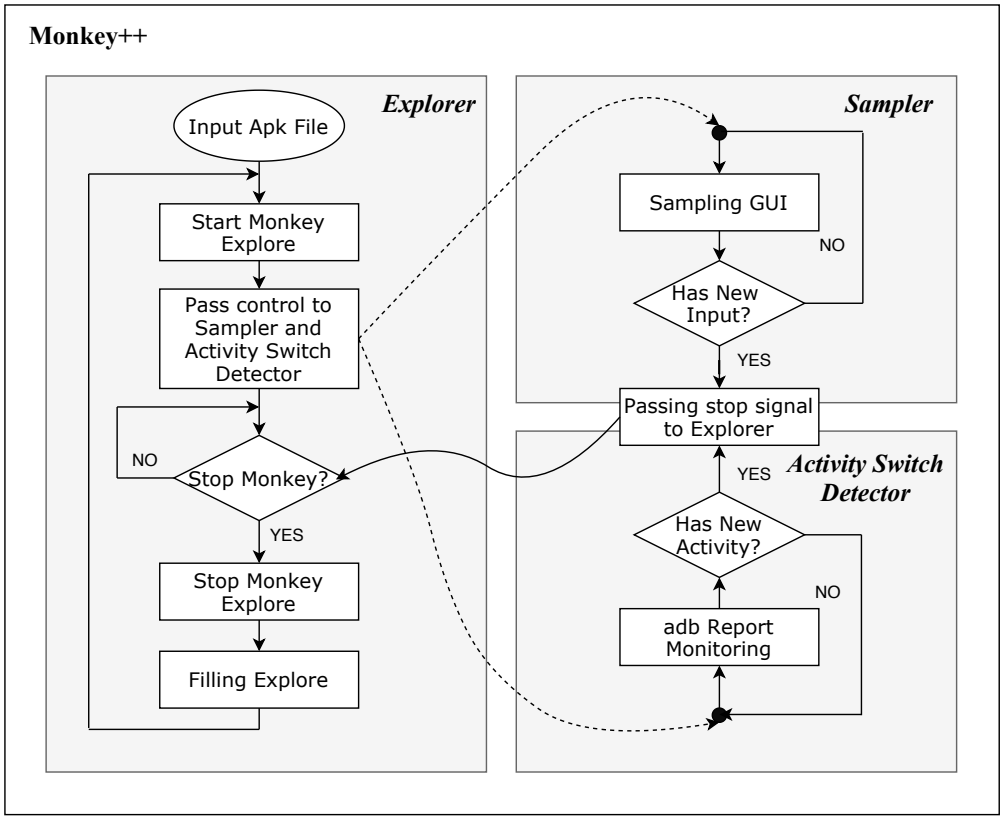


Figure 6.2: Monkey++ Overview

CHAPTER 7: SUMMARY CONCLUSION

We have answered the question of my thesis statement and confirmed the enhancement that the GUI understanding could bring in the analysis and exploration in mobile apps. We proved so by solving the problems in privacy security and improving the UI testing reusing from iOS to Android.

In the first work, we proposed a novel approach named GUILeak to detect privacy policy violations due to leak of user input data. To address the two technical challenges (infinite mapping and various GUI implementation), we adapted the GATOR framework, and developed hierarchical-mapping-based violation detection. We apply our approach on three important domains (finance, health and dating) and detected 21 strong violations and 18 weak violations in 120 popular apps from the domains. Our experiment shows that our best technique variant can achieve a F score of 84% with proper similarity threshold set. In the second work, we propose a novel approach name TestMig that migrates iOS UI test cases to Android UI test cases. Specifically, we record the iOS UI event sequences invoked during the iOS UI testing, and use the sequence transduction technique to convert this sequence to a sequence of Android UI events. Then, we explore the Android version of the software with the Android UI events and record the test cases. We evaluate our approach on four popular cross platform mobile apps, and the result shows that our approach can successfully convert averagely 80.2% of the iOS UI test cases to Android test cases, and achieve an average test coverage of 59.7%, which is close to the 60.4% average test coverage of the original iOS test cases on the iOS versions of the apps.

And in the future, I will continually work on GUI understanding and provide a better analysis and exploration for GUI testing and many more.

BIBLIOGRAPHY

[1] Android Monkey. https://developer.android.com/studio/test/monkey.

[2] Google play statistics, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2017-08-23.

[3] International data corporation (idc) smartphone os market share 2017 q1, http://www.idc.com/promo/smartphone-market-share/os. Accessed: 2017-08-23.

[4] Mint by the numbers: Which user are you?, https://blog.mint.com/credit/mint-by-the-numbers-which-user-are-you-040616/. Accessed: 2017-08-23.

[5] Sougou. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogouhl=en. Accessed: 2014-08-30.

[6] Ui privacy project web site, https://sites.google.com/site/uiprivacy2017/. Accessed: 2017-02-22.

[7] Android, http://developer.android.com/guide/topics/ui/index.html, 2013.

[8] Apple app store, http://www.apple.com/itunes/charts/, 2013.

[9] Bing translator, http://www.bing.com/translator, 2013.

[10] Cocoa touch, https://developer.apple.com/technologies/ios/cocoa-touch.html, 2013.

[11] Flappy bird. https://en.wikipedia.org/wiki/Flappy_Bird, 2013.

[12] Github, http://github.com/, 2013.

[13] Google translate, http://translate.google.com/, 2013.

[14] Java swing, http://docs.oracle.com/javase/tutorial/uiswing/, 2013.

[15] Kbabel, http://l10n.kde.org/tools/, 2013.

[16] Microsoft.net, http://www.microsoft.com/net, 2013.

[17] Open office, http://www.openoffice.org/download/other.html, 2013.

[18] Passolo, http://www.sdl.com/products/sdl-passolo/, 2013.

[19] Poedit, http://www.poedit.net/, 2013.

[20] Sisulizer, http://www.sisulizer.com/, 2013.

[21] Sourceforge, http://sourceforge.net/, 2013.

[22] W3c dom, http://www.w3.org/dom/, 2013.

[23] iOS torches Android when it comes to developer profits. http://bgr.com/2016/07/20/ios-vs-android-developers-profits-app-store-google-play/, 2016.

[24] Apache cordova. https://cordova.apache.org/, 2017.

[25] Clean master. https://play.google.com/store/apps/details? id=com.cleanmaster.mguard, 2017.

[26] Cordova vs. native apps. http://mubaloo.com/cordovavsnativeapps/, 2017.

[27] Crossy road. https://play.google.com/store/apps/details? id=com.yodo1.crossyroad, 2017.

[28] Facebook. https://www.facebook.com/, 2017.

[29] Facebook messager. https://www.messenger.com/, 2017.

[30] Instagram. https://www.instagram.com/, 2017.

[31] Jacoco. http://www.eclemma.org/jacoco/, 2017.

[32] Kika emoji. https://play.google.com/store/apps/details? id=com.qisiemoji.inputmethod, 2017.

[33] Monkey runner. https://developer.android.com/studio/test/ monkeyrunner/index.html, 2017.

[34] Pandora radio. https://www.pandora.com/, 2017.

[35] Snapchat. https://www.snapchat.com/, 2017.

[36] Spotify. https://www.spotify.com/, 2017.

[37] Ui automator. https://developer.android.com/training/testing/ui-automator.html, 2017.

[38] Unity 3d. https://unity3d.com/, 2017.

[39] Whatsapp. https://www.whatsapp.com/, 2017.

[40] Appium, http://appium.io/, 2018.

[41] The xcode ide. https://developer.apple.com/xcode/, 2018.

[42] Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Proc. ICPC*, pages 156–159, 2010.

[43] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[44] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261, 2012.

[45] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[46] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, 32(5):53–59, 2015.

[47] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[48] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[49] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proc. IWPSE*, pages 31–40, 2004.

[50] Laura Arjona Reina and Gregorio Robles. Mining for localization in android. In *Proceedings of Internation Working Conference on Mining Software Repositories*, pages 136–139, 2012.

[51] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, Jun 2014.

[52] Masayuki Asahara and Yuji Matsumoto. Extended models and tools for high-performance part-of-speech. In *Proc. COLING*, pages 21–27, 2000.

[53] Mikhail Auguston, James Bret Michael, and Man-Tak Shing. Environment behavior models for scenario generation and testing automation. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, 2005.

[54] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[55] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 641–660, 2013.

[56] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. OOPSLA*, pages 265–279, 2005.

[57] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 265–279, 2005.

[58] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289, 2013.

[59] G. Bavota, M. Linares-Vasquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, (99):1–1, 2014.

[60] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.

[61] Harvey Russell Bernard. *Research methods in anthropology: Qualitative and quantitative approaches*. Rowman Altamira, 2011.

[62] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[63] Jaspreet Bhatia and Travis D Breaux. Towards an information type lexicon for privacy policies. In *Requirements Engineering and Law (RELAW), 2015 IEEE Eighth International Workshop on*, pages 19–24. IEEE, 2015.

[64] John Blatz, Erin Fitzgerald, George Foster, Simona Gandrabur, Cyril Goutte, Alex Kulesza, Alberto Sanchis, and Nicola Ueffing. Confidence estimation for machine translation. In *Proceedings of the 20th International Conference on Computational Linguistics*, 2004.

[65] S.A. Bohner. Impact analysis in the software change process: a year 2000 perspective. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 42–51, 1996.

[66] Mitra Bokaei Hosseini, Xue Qin, Xiaoyin Wang, and Jianwei Niu. Extracting information types from android layout code using sequence to sequence learning. *AAAI Workshop on NLP for Software Engineering*, Feb 2018.

[67] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.

[68] Johannes Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.

[69] Travis D Breaux and Florian Schaub. Scaling requirements extraction to the crowd: Experiments with privacy policies. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 163–172. IEEE, 2014.

[70] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9–29, 2005.

[71] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130, 2011.

[72] Thomas E. Cheatham, G.H. Holloway, and J.A. Townley. Symbolic evaluation and the analysis of programs. *Software Engineering, IEEE Transactions on*, SE-5(4):402–417, July 1979.

[73] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar, October 2014. Association for Computational Linguistics.

[74] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *Network and Distributed System Security Symposium*, 2013.

[75] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 141–152, 2009.

[76] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

[77] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[78] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proc. ICSM*, pages 359–368, 1996.

[79] Kingsum Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 359–368, 1996.

[80] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[81] Malcolm Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

[82] J. Clarke. Automated test generation from a behavioral model. In *Proceedings of the Pacific Northwest Software Quality Conference*, 1998.

[83] Michael J. Collins and Terry Koo. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70, March 2005.

[84] Federal Trade Commission. Federal Trade Commission Act, 2010. Public Law 111-203.

[85] Senate Banking Committee. Gramm-leach-bliley act, 1999. Public Law 106-102.

[86] Bradley Cossette and Robert J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 55, 2012.

[87] PCI Security Standards Council. Payment card industry data security standard, 2016. version 3.2.

[88] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

[90] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. ICSE*, pages 481–490, 2008.

[91] V. Dagiene and R. Laucius. Internationalization of open source software: framework and some issues. In *2nd International Conference on Information Technology: Research and Education*, pages 204–207, 2004.

[92] Shuaifu Dai, A. Tongaonkar, Xiaoyin Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*, pages 809–817, 2013.

[93] Shuaifu Dai, A. Tongaonkar, Xiaoyin Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*, pages 809–817, 2013.

[94] A. Danko. Formalization of functional aspects in business software globalization. In *14th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, pages 107–116, 2010.

[95] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *IN PROC. INTâL CONF. ON LANGUAGE RESOURCES AND EVALUATION (LREC*, pages 449–454, 2006.

[96] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. OOPSLA*, pages 166–177, 2000.

[97] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proc. ECOOP*, pages 404–428, 2006.

[98] D. Dig and R. Johnson. The role of refactorings in api evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 389–398, 2005.

[99] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *JSME*, 18(2):83–107, March 2006.

[100] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. ICSE*, pages 427–436, 2007.

[101] Christian Fritz (ec Spride, Steven Arzt (ec Spride, Siegfried Rasthofer (ec Spride, Eric Bodden (ec Spride, Alexandre Bartel, Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Re Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Mcdaniel. Damien octeau (penn state university) patrick mcdaniel (penn state university) highly precise taint analysis for android application, 2013.

[102] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, March 2003.

[103] Xiaoyin Wang Eric Ruiz, Shaikh Mostafa. Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries. In *Technical Report*, 2015.

[104] T. Espinha, A. Zaidman, and H.-G. Gross. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 84–93, 2014.

[105] B. Esselink. *A Practical Guide to Software Localization: For Translators, Engineers and Project Managers*. John Benjamins Publishing Co, 2000.

[106] European Parliament and Council. General Data Protection Regulation, 2016. Regulation (EU) 2016/679.

[107] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Applications of linguistic techniques for use case analysis. *Requirement Engineering*, 8(3):161–170, March 2003.

[108] Peter Gerstl and Simone Pribbenow. A conceptual theory of part-whole relations and its applications. *Data & Knowledge Engineering*, 20(3):305–322, 1996.

[109] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56, 2010.

[110] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, February 2005.

[111] Peter GrÂÂnwald. A minimum description length approach to grammar inference. In *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, volume 1040 of *Lecture Notes in Computer Science*, pages 203–216. Springer Berlin Heidelberg, 1996.

[112] Elvis Hau and Manuela Aparício. Software internationalization and localization in web based erp. In *Proceedings of the 26th Annual ACM International Conference on Design of Communication*, SIGDOC '08, pages 175–180, 2008.

[113] Z. He, D. W. Bustard, and X. Liu. Software internationalisation and localisation: Practice and evolution. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming and the Second Workshop on Intermediate Representation Engineering for Virtual Machines*, pages 89–94, 2002.

[114] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *Proc. ICSE*, pages 274–283, 2005.

[115] Frank Annunzio Henry Reuss. Right to financial privacy act, 1978. Public Law 95-630.

[116] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[117] James H. Hogan, Chris Ho-Stuart, and Binh Pham. Current issues in software internationalisation. In *Proc. Australian Computer Science Conference*, pages 1–10, 2003.

[118] Mitra Bokaei Hosseini, Sudarshan Wadkar, Travis D Breaux, and Jianwei Niu. Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series*, 2016.

[119] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 977–992, Berkeley, CA, USA, 2015. USENIX Association.

[120] Jianjun Huang, Xiangyu Zhang, and Lin Tan. Detecting sensitive data disclosure via bidirectional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 169–180, New York, NY, USA, 2016. ACM.

[121] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 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, pages 1036–1046, New York, NY, USA, 2014. ACM.

[122] W. J. Hutchins and H. L. Somers. Academic Press, 1992.

[123] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, 2005.

[124] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, 2002.

[125] Mona E. Joorabchi, Mohamed Ali, and Ali Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 450–460, 2015.

[126] C. Kemper and C. Overbeck. What's new with JBuilder. In *Proc. JavaOne*, 2005.

[127] Iyad Khaddam and Jean Vanderdonckt. Flippable user interfaces for internationalization. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 223–228, 2011.

[128] Adam Kilgarriff and Christiane Fellbaum. Wordnet: An electronic lexical database, 2000.

[129] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811, 2013.

[130] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. ICSE*, pages 333–343, 2007.

[131] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proc. WCRE*, pages 143–152, 2005.

[132] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[133] Dan Klein and Christopher D. Manning. Fast exact inference with a factored model for natural language parsing. In *In Advances in Neural Information Processing Systems 15 (NIPS*, pages 3–10. MIT Press, 2003.

[134] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *Proc. RE*, pages 121–130, 2007.

[135] S. Kokkots and Constantine D. Spyropoulos. An architecture for designing internationalized software. In *Eighth IEEE International Workshop on Software Technology and Engineering Practice*, pages 13–21, 1997.

[136] Z. Kozareva, Ó. Ferrández, A. Montoyo, Rafael Muñoz, Armando Suárez, and Jaime Gómez. Combining data-driven systems for improving named entity recognition. *DKE*, 61(3):449–466, June 2007.

[137] Paul Krebs and T. Dustin Duncan. Health app use among us mobile phone owners: A national survey. *JMIR mHealth uHealth*, 3(4):e101, Nov 2015.

[138] H. P. Krings, G. S. Koby, G. M. Shreve, K. Mischerikow, and S. Litzer. Kent State Univeristy Pr, 2001.

[139] A. Kull. Automatic gui model generation: State of the art. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, pages 207–212, 2012.

[140] ShuvenduK. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique RebÃªlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717, 2012.

[141] Leslie Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[142] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13, 2012.

[143] Wee Kheng Leow, Siau Cheng Khoo, and Yi Sun. Automated generation of test programs from closed specifications of classes and test cases. In *Proceedings of the 26th International Conference on Software Engineering*, pages 96–105, 2004.

[144] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 477–487, 2013.

[145] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 643–653. IEEE, 2017.

[146] Fan Long and Martin Rinard. Staged program repair in spr. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2015.

[147] Nuno G. Lopes and Carlos J. Costa. Erp localization: Exploratory study in translation: European and brazilian portuguese. In *Proceedings of the 26th Annual ACM International Conference on Design of Communication*, SIGDOC '08, pages 93–98, 2008.

[148] Adam Lopez. Statistical machine translation. *ACM Comput. Surv.*, 40(3):8:1–8:49, 2008.

[149] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[150] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *ASEJ*, 10(2):183–202, April 2003.

[151] Aaron Marcus and Emilie West Gould. Crosscurrents: Cultural dimensions and global web user-interface design. *interactions*, 7(4):32–46, July 2000.

[152] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 287–296, 2003.

[153] Stephen McCamant and MichaelD. Ernst. Early identification of incompatibilities in multi-component upgrades. In *European Conference on Object-Oriented Programming*, pages 440–464, 2004.

[154] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, 2013.

[155] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 260–269, 2003.

[156] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[157] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–342, 2011.

[158] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511, 2013.

[159] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 353–363, Piscataway, NJ, USA, 2012. IEEE Press.

[160] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, pages 432–441, 2005.

[161] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, June 1997.

[162] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, 2014.

[163] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 466–476, 2013.

[164] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 993–1008, Berkeley, CA, USA, 2015. USENIX Association.

[165] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 457–468, 2014.

[166] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.

[167] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.

[168] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proc. OOPSLA*, pages 302–321, 2010.

[169] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, 2013.

[170] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[171] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318, 2002.

[172] Rajesh Parekh and Vasant Honavar. Grammar inference, automata induction, and language acquisition. In *Handbook of Natural Language Processing*, pages 727–764. Marcel Dekker, 2000.

[173] Wenlin Peng, Xiaohu Yang, and Feng Zhu. Automation technique of software internationalization and localization based on lexical analysis. In *Proceedings of the 2Nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 970–975, 2009.

[174] Manuel A. Pérez-Quiñones, Olga I. Padilla-Falto, and Kathleen McDevitt. Automatic language translation for user interfaces. In *Proceedings of the 2005 Conference on Diversity in Computing*, pages 60–63, 2005.

[175] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[176] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.

[177] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[178] Martin F Porter. Snowball: A language for stemming algorithms, 2001.

[179] Leo Postman and Laura W Phillips. Short-term temporal changes in free recall. *Quarterly journal of experimental psychology*, 17(2):132–138, 1965.

[180] X. Qin, S. Holla, L. Huang, L. Montijo, D. Aguirre, and X. Wang. How does machine translated user interface affect user experience? a study on android apps. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 430–435, 2017.

[181] Xue Qin, Hao Zhong, and Xiaoyin Wang. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 284–295, New York, NY, USA, 2019. Association for Computing Machinery.

[182] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387, 2012.

[183] Siegfried Rasthofer, Steven Arzt, Ec Spride, Technische UniversitÃt Darmstadt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks, Feb 2014.

[184] Xiaoxia Ren and Barbara G. Ryder. Heuristic ranking of java program edits for fault localization. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 239–249, 2007.

[185] Health Resources and Services Administration. Health insurance portability and accountability act, 1996. Public Law 104-191.

[186] Guido Röbetaling. Translator: a package for internationalization for java-based applications and guis. In *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, page 312, 2006.

[187] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 143:143–143:153, New York, NY, USA, 2014. ACM.

[188] S.L. Salas and Einar Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.

[189] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Guitar: Piecing together android app guis from memory images. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 120–132, New York, NY, USA, 2015. ACM.

[190] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[191] P. Sawyer, P. Rayson, and R. Garside. REVERE: Support for requirements synthesis from documents. *Information Systems Frontiers*, 4(3):343–353, March 2002.

[192] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proc. ICSE*, pages 471–480, 2008.

[193] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[194] D. Shepherd, Z. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. AOSD*, pages 212–224, 2007.

[195] Ahmed Shwan and Muhammad Murtaza. Master's thesis: Guidelines for multilingual software development. 2012.

[196] João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva, and José Creissac Campos. The guisurfer tool: Towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 181–186, 2010.

[197] Robert Simmons. *Hardcore Java*. O'Reilly Media, 2009.

[198] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 25–36, New York, NY, USA, 2016. ACM.

[199] Jonathan Slocum. A survey of machine translation: Its history, current status, and future prospects. *Comput. Linguist.*, 11(1):1–17, 1985.

[200] E. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting of the European Software Engineering Con-*

ference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2015.

[201] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing With Compositional Vector Grammars. In *ACL*. 2013.

[202] Nicholas A. Solter and Scott J. Kleper. *Professional C++*. Wrox, 2005.

[203] Harold Somers. Review article: Example-based machine translation. *Machine Translation*, 14(2):113–157, 1999.

[204] S. Staiger. Reverse engineering of graphical user interfaces using static analyses. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 189–198, 2007.

[205] D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, and Jeff Cogswell. *C++ Cookbook*. O'Reilly Media, 2005.

[206] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proc. OOPSLA*, pages 268–285, 1996.

[207] Hunt T. Cost effective software internationalisation. 17:1–6, 2013.

[208] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proc. SOSP*, pages 145–158, 2007.

[209] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proc. ICSE*, pages 11–20, 2011.

[210] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *International Conference on Software Engineering*, 2015.

[211] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proc. ASE*, pages 377–380, 2007.

[212] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 83–95, 2015.

[213] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: A human study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[214] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. ASE*, pages 327–336, 2008.

[215] Ferhan Ture, Douglas W. Oard, and Philip Resnik. Encouraging consistent translation choices. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 417–426, 2012.

[216] Nicola Ueffing and Hermann Ney. Word-level confidence estimation for machine translation. *Comput. Linguist.*, 33(1):9–40, 2007.

[217] E. Uren, R. Howard, and T. Perinotti. *Software Internationalization and Localization, An Introduction*. Van Nostrand Reinhold, 1993.

[218] N. Walkinshaw, M. Roper, and M. Wood. Feature location and extraction using landmarks and barriers. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 54–63, 2007.

[219] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[220] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of android test generation tools in industrial cases. In *Proceedings*

of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, page 738–748, New York, NY, USA, 2018. Association for Computing Machinery.

[221] X. Wang, X. Qin, M. Bokaei Hosseini, R. Slavin, T. D. Breaux, and J. Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 37–47, 2018.

[222] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, September 20-24, 2010*, pages 457–466, 2010.

[223] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 199–210, 2015.

[224] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proc. ISSTA*, pages 199–210. ACM, 2015.

[225] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-translate constant strings for software internationalization. In *International Conference on Software Engineering*, pages 353–363, 2009.

[226] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-translate constant strings in web applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 87–96, 2010.

[227] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis. *Software Engineering, IEEE Transactions on*, 39(4):516–536, 2013.

[228] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[229] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 32–41, 2007.

[230] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

[231] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 231–240, Sept 2006.

[232] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. ASE*, pages 231–240, 2006.

[233] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[234] L. White and H. Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 110–123, 2000.

[235] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 185–195, New York, NY, USA, 2016. ACM.

[236] Wei Wu, B. Adams, Y.-G. Gueheneuc, and G. Antoniol. Acua: Api change and usage auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 89–94, 2014.

[237] Wei Wu, Y. Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. ICSE*, pages 325–334, 2010.

[238] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[239] Xin Xia, D. Lo, Feng Zhu, Xinyu Wang, and Bo Zhou. Software internationalization and localization: An industrial experience. In *18th International Conference on Engineering of Complex Computer Systems*, pages 222–231, 2013.

[240] Shaikh Mostafa Xiaoyin Wang. Autobuilder: Towards automatic building of java projects to support analysis of software repositories. In *Technical Report*, 2014.

[241] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE TSE*, 33(12):818–836, December 2007.

[242] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for android (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 658–668, Nov 2015.

[243] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[244] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Automated Software Engineering*

(ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on, pages 20–29, Sept 2012.

[245] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, pages 183–200, 2002.

[246] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 253–267. 1999.

[247] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pages 23–32, 2011.

[248] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 765–784, 2013.

[249] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 518–529, New York, NY, USA, 2015. ACM.

[250] Xiangyu Zhang, R. Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 319–329, 2003.

[251] Ying Zhang and Bach Nguyen. Virtual babel: Towards context-aware machine translation in virtual worlds. In *Machine Translation Summit*, 2009.

[252] Hao Zhong and Hong Mei. An empirical study on API usages. *IEEE Transaction on Software Engineering*, 2018.

[253] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proc. ICSE*, pages 195–204, 2010.

[254] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proc. ASE*, pages 307–318, 2009.

[255] Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 137–146, 2010.

[256] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *Network and Distributed System Security Symposium NDSS*, 2017.

[257] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 91–100, 2009.

VITA

Xue Qin is a Ph.D. candidate at the Department of Computer Science at the University of Texas at San Antonio. Her research spans explicitly over different domains, including software engineering, privacy security, software testing, and mobile analysis. She is going to start a new role as an Assitant Professor at Villanova University, and she plans to continually contribute to the research of GUI analysis.