An Empirical Study of i18n Collateral Changes and Bugs in GUIs of Android apps

Camilo Escobar-Velásquez, Michael Osorio-Riaño, Juan Dominguez-Osorio, Maria Arevalo, Mario Linares-Vásquez
Systems and Computing Engineering, Universidad de los Andes, Bogotá, Colombia
{ca.escobar2434, ms.osorio, jm.dominguez, mi.arevalo10, m.linaresv}@uniandes.edu.co

***Abstract*—Mobile markets allow developers to easily distribute mobile apps worldwide and collect complaints and feature requests in the form of user reviews and star ratings. Therefore, internationalization (*i18n*) of apps is a highly desired feature, which is currently supported in mobile platforms by using resources files with strings that can be internationalized manually. This manual translation can be a time consuming and error-prone task when the app is targeted for different languages and the amount of strings to be internationalized is large. Moreover, the lack of consideration of the impact of internationalized, strings can drive to collateral (*i.e.,* unexpected) changes and bugs in the GUI layout of apps.**

In this paper, we present an empirical study on how i18n can impact the GUIs of Android apps. In particular, we investigated the changes, bugs and bad practices related to GUIs when strings of a given default language (*i.e.,* English in this case) are translated to 7 different languages. To this, we created a source-codeless approach, `ITDroid`, for automatically (i) translating strings, and (ii) detecting bad practices and collateral changes introduced in the GUIs of Android apps after translation. `ITDroid` was used on a set of 31 Android apps and their translated versions. Then, we manually validated the i18n changes that introduced bugs into the GUIs of the translated apps. Based on these results, we present a taxonomy of i18n changes and bugs found along in the apps as well as implications of our findings for practitioners and researchers. Online appendix: https://thesoftwaredesignlab.github.io/ITDroid/

***Index Terms*—Internationalization, Android, collateral changes, bugs, bad practices**

I. Introduction

Mobile apps are a type of software application that has no comparative in terms of adoption in human daily activities, nowadays. The distribution model provided by online app markets has pushed users and developers towards a new dynamic in terms of release engineering practices, apps consumption in mobile devices, and requirements elicitation [1]–[3]. Mobile markets allow developers to easily distribute mobile apps worldwide and collect complaints and feature request in the form of user reviews and star ratings at an unprecedented rate. Therefore, mobile app developers should be more aware of existing practices for making their apps accessible worldwide, which includes internationalizing the apps, but, without impacting the quality as perceived by users, and without being a "show-stopper" for the development process.

Acknowledgment. Escobar-Velásquez and Linares-Vásquez are partially supported by a Google Latin American Research Award 2018-2020.

The term Internationalization (*i18n*) refers to mechanisms for adapting software applications to different languages. From the perspective of automated software engineering, internationalization-related tasks can be summarized in (i) detecting and translating strings that need to be internationalized, and (ii) testing that *i18n* efforts do not introduce bugs in the internationalized apps. Current practices for internationalizing software applications consist mostly on extracting typically hard-coded strings to resources files, and then, creating internationalized versions of the strings in the default language; it means, there should be a resource file for each language (included the default one), containing the texts that are displayed in the GUI. Mobile development IDEs provide wizard-based features for detecting hard-coded strings and generating (manually) the internationalized versions, however, there is no feature in the IDEs for automatically translating the strings. Recent approaches proposed by Wang *et al.* [4]–[7] automatically detect need-to-translate strings, and translate the strings but in an off-line mode, *i.e.,* the strings are translated outside and not incorporated into the app.

On the other side, automated *i18n* testing of mobile apps has not been widely explored. Automated testing of mobile apps have focused on functional bugs [8]–[12], performance issues [13], vulnerabilities [14], [15], among other type of bugs. In spite of those efforts, in the case of *i18n* bugs in Android apps, there is no previous study on this type of bugs and there is no available approach for automatically detecting the bugs on Android apps.

Previous efforts on detecting *i18n* issues has been done but on the domain of web applications [16]–[18], which provide interesting insights and learned lessons. However, transferring the proposed approaches to the domain of Android apps requires to consider the specifics of the Android programming model. Therefore, in this paper we first describe a novel approach (`ITDroid`) for automatically detecting *i18n* changes in Android apps in a source-codeless fashion (*i.e.,* without having access to the apps source code), and then, we analyze different aspects of *i18n* changes detected in a set of 31 Android apps and their automatically internationalized versions to 7 different languages. `ITDroid` combines APK static analysis, automated translation of strings, and dynamic analysis techniques (*i.e.,* GUI ripping and automated replay), to identify violations of GUI constraints when simulating apps execution with different languages. `ITDroid` also detects

strings hard coded and declared in resource and code files that are not internationalized. The proposed approach operates in a source-codeless fashions, thus, it is agnostic of the native language used for creating the app (*i.e.,*, Java and Kotlin) because the analysis is done at the APK level.

After analyzing 31 Android apps and the internationalized versions (which account for a total of 62 APK files) we found that all of the studied apps have *i18n* bugs introduced by (i) bad development practices such as hardcoded strings or non existence of a *strings.xml* file for a certain language, or (ii) *i18n* collateral changes. The collateral changes, bad practices, and bugs detected in the analyzed are depicted in a taxonomy. Beside the taxonomy, we discuss representative examples a set of implications for practitioners and researchers.

II. Software Internationalization

Software applications contain textual information that is displayed to users via command lines, GUI, exceptions, messages, logs, etc. When applications are expected to be deployed/delivered in different languages (*e.g.,* English, Spanish, Chinese), *internationalization (i18n)* and *localization (l10n)* mechanisms should be included in the apps, with the purpose of displaying the textual information in the language of the host device (*e.g.,* a mobile device or a web browser running on a laptop) without installing additional software. While *i18n* is focused on adapting textual information to different languages and regions, *l10n* focuses on the specific requirements of a particular region. Note that in this paper we focus only on *i18n* when using English as the default language of Android apps, and seven target languages for the translation: (i) Italian, (ii) French, (iii) Russian, (iv) Arabic, (v) Spanish, (vi) Hindi and (vii) Portuguese.

A general and widely used mechanism for internationalizing software applications, independently of the type of app, consists on using textual files — one for each of the target languages — with key-value pairs for the internationalized strings. In those files, a key is the identifier for referencing a string from the app code, and the value is the corresponding literal that will be displayed for a host language. For example, in the case of Android apps, an internationalized string definition in English looks like this:

```
1  <string name="login_cancel_button">Cancel</string>
```

For Android apps, the internationalized strings are located in XML files stored at the `/res` folder of an app bundle. There should be a folder, for each language, containing the corresponding `strings.xml` file, *e.g.,* `/res/values-fr/string.xml` for French (`fr`) and `/res/values-ja/string.xml` for Japanese (`ja`). The Android Studio IDE includes wizards for extracting hard-coded strings to the *i18n* files, and for manually defining the different translations. However, despite having native support for *i18n* at the framework/API levels, when no automated support is available for strings translation, this is an error prone and potentially expensive task [19], [20]. Therefore, one way to manually deal with strings internationalization, widely used by open source projects, consist of delegating the translation task to global contributors that commit their own language files and translations.

A. Automated Support

Previous approaches have been proposed for automatically detecting the strings that need to be internationalized in software applications, and for automatically translating the strings. For instance, Wang *et al.* [4]–[6] propose a string-taint-analysis-based approach for automatically locating strings literals that are displayed in the GUI of Java and web applications [21]. Concerning automated translation of strings to be internationalized, Wang *et al.* [7] used RNN (Recurrent Neural Network) encode-decoders to automatically translate text in Android apps from English to the other five United Nations official languages (*i.e.,* Arabic, Chinese, French, Spanish and Russian). Note that Wang *et al.* [7] extracted the strings directly from the `string.xml` file of each analyzed app and did not generate new versions of the apps with the translations (*i.e.,* the translations were done off-line).

Other approaches have focused on internationalizing source code, but, from the perspective of transforming code to be Unicode compatible. For example, Xia *et al.* [22] proposed an approach for automatically transforming code, in a large legacy commercial system written in C/C++, in order to support the Unicode standard.

B. Internationalization Failures

Previous studies have shown that internationalization efforts can introduce Internationalization Failures (IFs) in web apps [16]–[18]. There are two types of IFs reported by [17]: Layout Failures (*a.k.a.,* Internationalization Presentation Failures- IPFs [16]), and Configuration Failures. The former are unintended modifications to GUI layouts, which are introduced because text translations can have different lengths, heights (when compared to the original text) and directions as in the case of right-to-left languages. For instance, while the word “car” in English has three characters, the translated word in Russian has 10 characters. Examples of these bugs are components overlapping, texts going beyond the container margins, and components that disappear from users view. On the other side, configuration failures are issues in the configuration of an *i18n* related property in a website or a web server [17].

A previous study on web applications, by Alaamer and Halfond [17], showed that IPFs are common on websites and they depend on specific languages. In terms of existing approaches for detecting IPF failures, Alameer *et al.* [16], [17] used layout graphs to detect presentation differences between the GUI layout graph of an app using the original language and the graph of an internationalized version. A layout graph describes the location, alignment, and overflow relationships that exist between components of a given GUI. An IPF is detected when there are differences between the original graph and the internationalized version. With the same purpose of

detecting general presentation failures (*i.e.,* not only IPFs) Mahajan and Halfond [23] used computer vision techniques to detect differences between GUI screenshots.

While the previously mentioned papers focused on detecting and locating the IPFs in the GUI, Mahajan *et al.* [18] automatically repair IPFs in websites by using search-based techniques that provide solutions as sets of HTML and CSS statements that fix the IPFs. This approach was able to fix all the issues in 18 out of 23 websites. However, note that the aforementioned approaches for detecting and fixing IPFs in web apps do not support automated exploration of the apps under analysis, it means, the research/user manually provides links to individual web pages (original and internationalized ones); in addition, the aforementioned approaches do not automatically translate the original version of the app under analysis to the target languages, which means that those approaches are highly dependent on the availability of the internationalized versions of the app.

C. Layout Testing of Mobile apps

It is worth noting that for the case of mobile apps, there are few works aimed at automatically supporting apps' internationalization. As previously mentioned, Wang *et al.* [7] provide off-line translation using RNNs. Concerning tools created by the platform designers (*i.e.,* Google and Apple), there is no tool or approach for automated translation of strings or IPFs detection/repair. The only available official support is for hard-coded strings detection and manual definition of internationalized strings in both the Android Studio and the iOS Xcode IDEs.

A common technique used by practitioners during internationalization/localization testing is called "pseudo-localization" which consists of replacing original strings with dummy text or random strings that follow pseudo-languages, *i.e.,* pre-defined heuristics such as using strings with double length or written from righ-to-left [24] For example, the iOS XCode IDE allows for testing iOS apps with pseudo-languages and right-to-left layouts. However, testing needs to be done manually or with unit tests created by developers. Another existing tool is LayoutTest by Linkedin [24] which allows for writing unit tests that check layout changes in iOS apps; the mock data used in the tests need to be manually defined by the developer.

Summary. Current approaches for automated support of internationalization have focused on five tasks [16], [17], [22]: (i) detection/location of strings that need to be translated, (ii) extraction of hard-code strings to resource files, (iii) translation of strings to target languages, (iv) automated detection/location of IPFs, and (v) automated fixing of IPFs. However support for internationalization tasks in Android apps is limited to off-line translation, detection of non-internationalized strings, and unit testing of layouts with mock data or pseudo-languages. Nowadays, there is no comprehensive approach, for web or Android apps, able to automatically (i) detect strings to be internationalized, (ii) create internationalized versions of the app under analysis, (iii) explore the app to build layout graphs of the apps, and (iv) detect *i18n* related changes in the GUIs. In addition, to the best of our knowledge, there is no empirical study aimed at reporting the amount and types of *i18n* changes and bugs in Android apps.

III. Detecting i18n Collateral Changes in Android Apps

With the purpose of providing developers and researchers with an automatic way to detect *i18n* changes in Android apps, in this section we describe the architecture and workflow of our `ITDroid` approach. `ITDroid` aims at:

1) Detecting non-internationalized strings (in source code and resource files),
2) Generating internationalized versions of an app under analysis,
3) Exploring the original version of an APK to generate a reference collection of GUI states (represented as a tree of layout graphs) and a replayable scenario of the execution,
4) Replaying the exploration scenario while using the app with different languages, and
5) Reporting internationalization (*i18n*) changes found on the internationalized versions.

`ITDroid` was designed to work with Android Packages files (APKs), which enables analysis when no source code is available, *i.e.,* `ITDroid` works on apps created with different native languages (*i.e.,* Java and Kotlin). Working with APK files instead of source code, reduces time overload when compiling source code to APK files.

Different than using mock-data or pseudo-languages, our approach automatically translates non-internationalized strings to a list L of targeted languages.

Fig. 1 depicts the `ITDroid` architecture and workflow. Note that the whole workflow is automated. We combine different techniques such as static analysis of APK files, automated GUI ripping and replay, and (*i18n*) changes detection. We describe each one of the techniques and components as follows.

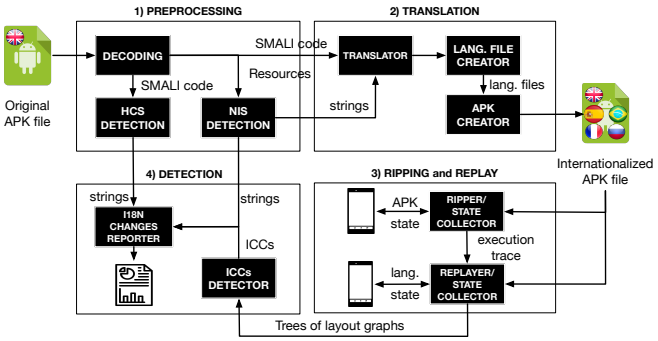


Fig. 1: `ITDroid` architecture and workflow

A. Preprocessing

In order to obtain a representation of the APK under analysis, we rely on the SMALI representation that can be extracted from APK files. Our choice for SMALI is because it has been recognized as one of the top representations used for

static analysis of APKs [25], [26]. ITDroid uses the apktool [27] library to extract the decoded resources and a SMALI representation of the dex code.

By using static analysis over the processed APK, ITDroid locates all the hardcoded strings. In SMALI, string literals are declared via const-string instructions (see Snippet 1). Therefore, ITDroid generates the AST of the SMALI representation and by using a visitor pattern, it goes through all nodes looking for instructions that match the desired expression. After all hardcoded strings (HCS) are located, those are reported, and grouped by method and class. It is worth noticing that hardcoded strings are not translated.

Snippet 1: Hardcoded String Example

```
1   const-string v1, "mileage-export"
```

Additionally, ITDroid identifies strings in the resources of the app that are not internationalized. ITDroid compares the string.xml file of the default language (*i.e.,* English) with the existing files (if any) of each target language. It is worth noting that there is a set of strings automatically added and translated by the Android SDK, therefore, ITDroid removes those strings from the analysis. The result is then a list of tuples $< lang, string\_id >$ that describes non-internationalized strings (NIS), with $lang$ being a target language in L, and $string\_id$ the id of an existing string in the default language but not internationalized to $lang$.

B. Translation

After identifying the non-internationalized strings in the app resources folder, ITDroid proceeds with the translation. By using a strategy pattern, ITDroid delegates the translation to an external engine through an interface. This design decision guarantees that ITDroid is agnostic to the translation engine. For our initial implementation, we consumed the IBM Watson Translation service, since it has a free and easy to use API.

Before doing the translation, ITDroid preprocesses the strings to replace tokens like *%s, %d*, which cause problems in the translation process. Once the strings in NIS are translated, the results are used to build corresponding string.xml files for each of the target languages in L. The files are included into the decoded original APK, and then repackaged into an internationalized APK (A^I). Note that the new version of the app is ready to be installed and tested using any of the selected languages.

C. Ripping & Replay

Our approach includes a regression testing-based approach for automatically identifying Internationalization Collateral Changes (ICCs) in an internationalized APK (A^I) . An Internationalization Collateral Change (ICC) is a change introduced in the GUI of a mobile app when the app is internationalized and executed in a device (physical or emulator) that is configured for a language different than the default app language. Note that Alameer *et al.* [16], [17] use the terms *layout failures* and *internationalization presentation failures* to describe presentation issues introduced in a GUI when internationalizing an application. However, in this paper we introduce the term Internationalization Collateral Changes (ICCs), because not all the induced changes produce bugs in a GUI. Therefore, we prefer to use *collateral change* instead of *failure* to describe changes induced in GUIs when using internationalized strings.

ITDroid collects GUI states in both A and A^I, assuring that (i) the same scenarios are executed automatically in both versions, (ii) the execution is independent of the existence of automated tests, and (iii) there is no need of human-collected replayable scenarios. Therefore, ITDroid automatically explores the GUI of A on an Android emulator, by following a DFS-based approach widely used in Android GUI rippers [28]–[31]. Our approach uses a systematic exploration algorithm that traverses an app GUI, extracting a snapshot of the different UI states and processing those to identify the elements that can be exercised in the current view. Since a view can have several changes due to the execution of events on its elements, ITDroid uses a state-based execution that stores the characteristics of the current state of the view and generates a set of unique states while exploring the app. Because of this, ITDroid identifies the events that trigger new states during the exploration. Consequently, ITDroid reports the sequence of events that were generated during the app exploration.

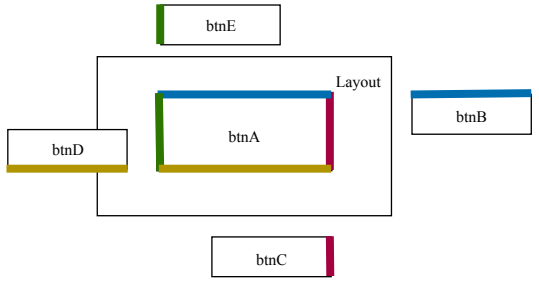
Additionally, to avoid manually exploring the internationalized APK, our ripper implementation replays the sequence of events, generated on the original APK, on emulators automatically configured by ITDroid to use the target languages. Therefore, the internationalized APK is explored seven times, one time for each target language

Since the set of steps that produce a change of state are already identified from the original exploration, when replaying the exploration, our tool does not execute all the intermediate events that did not trigger a new state; this fast replay that avoid useless events, allows ITDroid to improve the time required to explore an internationalized version of the app. During the ripping, the GUI states are stored in a tree structure, where each node represents a GUI state, and the edges are transitions that lead to different GUI states. Each node in the tree stores the state id, a screenshot of the GUI, and the list of GUI components. It is worth noticing that while replaying events, we ignore the text content of the GUI elements, to avoid miss-identification of elements due to the language change in the device.

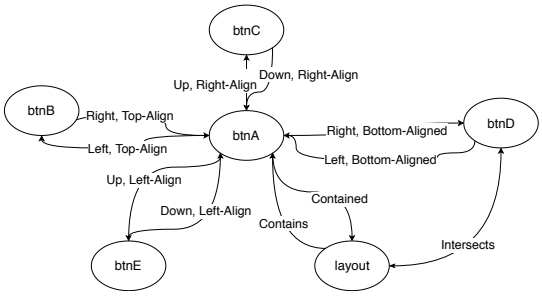
To avoid reproducibility issues, we assure a cold-start scenario, *i.e.,* before a each execution (either ripping o replay) the emulator is restarted, the app is always un-installed/installed and its local data is deleted. Also, ITDroid does not generate replayable steps that are coupled to components locations, which can lead to missing steps; we instead rely on locating the components via a composed id (xpath, and element id).

For identifying ICCs, ITDroid models the states by using the layout graph representation proposed by Alameer *et al.* [16], [17]. A layout graph (TLG) describes the existing relationships between all components of a given GUI, in terms of **location** (*i.e.,* up, down, right, left), **alignment** (*i.e.,* top-, bottom-, right-, left-aligned), and **overlapping** (*i.e.,* contains, contained, intersects).

Fig. 2 serves as an example for explaining layout graphs. In the example (see Fig. 2a) there are 5 buttons and 1 layout component: *btnA* is located at the center of the image and for the purpose of this example, we are going to analyze the relations between this button and the rest of the elements. *btnA* has **location** relations: *btnE* is **up**, *btnB* is to the **right**, *btnD* is to the **left**, and *btnC* is **down**. In terms of **alignment**, *btnE* is **left-aligned**, *btnB* is **top-aligned**, *btnD* is **bottom-aligned** and *btnC* is **right-aligned**. Each of these alignment relations is shown using a different color. Finally, in terms of **overlapping**, there is a **containment** relation with the *Layout* component. As the reader can see in Fig. 2a there is also a relation between *btnD* and *Layout* called **intersection**. Fig. 2b presents the layout graph ([16], [17]) depicting the relations. Note that the graph is bidirectional and the relations can be different under each direction, except for the case of **intersection**.



(a) GUI components



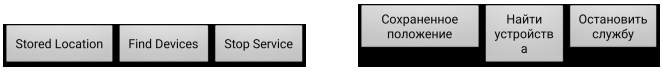
(b) Layout graph

Fig. 2: Example of a layout graph: (a) view of GUI components distribution, and (b) corresponding layout graph.

D. Detection and Reporting

Once the app has been explored for each language in L, the next step is to process the corresponding layout graphs. Therefore, using the default language' layout graph (TLG_{df}) as a baseline, `ITDroid` goes through each language's layout graph (TLG_{lang}, $\forall lang \in L$) looking for differences with the baseline graph. `ITDroid` starts by trying to find a matching node in TLG_{lang} for each one in TLG_{df}. Once all the possible nodes are paired, the next step is to recognize the relations that have changed between the two graphs. The result of this step is the set of relations added and lost for all pair of nodes in each state. Those changes are then detected as Internationalization Collateral Changes (ICCs).

Fig. 3 shows the English (a) and Russian (b) version of a set of buttons from the *a2dp.Vol* app' main activity. Each image contains 4 Android components: 3 buttons and a *linear layout*. In the English version (Fig. 3a) all the components are bottom-aligned –taking into account buttons margin. However, for the Russian version (Fig. 3b) only the first and third buttons remain bottom-aligned. Therefore, from the point of view of the left-most button in the Russian version, there are 2 relations that were lost: bottom alignment with the second button and with linear layout.



(a) English version (b) Russian version

Fig. 3: Example of an ICC

At the end of the process, `ITDroid` generates a report listing (i) hard-coded strings, (ii) non-internationalized strings declared in the default language but not traduced in the target languages, and (iii) internationalization collateral changes. To easily handle ICCs, `ITDroid` reports the changes by identifying: (i) the exploration state id, (ii) the id of the node, (iii) each node for which a relation was modified along with the details of the modified relations, and (iv) screenshots depicting the GUI state for both default and internationalized languages.

IV. Empirical Study Design

As of today, there is no empirical study analyzing *i18n* bugs in Android apps. Therefore, we used `ITDroid` in an empirical study aimed at analyzing different aspects of *i18n* changes in Android apps. To this, we executed `ITDroid` on the APK files of 31 open source Android apps having English as their default language, with the purpose of answering the following research questions (RQs):

- **RQ$_1$:** *What types of i18n collateral changes and bugs are exhibited on GUIs of Android apps?*
- **RQ$_2$:** *What i18n collateral changes induce i18n collateral bugs on GUIs of Android apps?*
- **RQ$_3$:** *What are the GUI components and languages more prone to i18n collateral changes and bugs?*

A. Context of the study

To select the dataset of apps for this study, we started looking for Android apps used in studies aiming at evaluating testing approaches: DroidMate [32], CrawlDroid [33], MDroid+ [34], [35]). Thus, we created a dataset of 211 apps including the APKs used in those studies, and an internal dataset we built for previous studies. Afterwards, to evaluate whether the apps were valid for our study, we followed these inclusion/exclusion criteria:

- We discarded the apps that were not available at the Google Play Store.
- Given the fact that we were going to execute the apps on an API 27 Android Emulator, we removed apps that were not compatible with the API 27. Note that our choice for

running the experiments on an emulator rather than on a physical device is because ITDroid is tailored for working with emulators, for instance, ITDroid automatically sets the emulator language via ADB commands.
- We included (i) apps with more than one activity, or (ii) apps with only one activity but exhibiting more than one GUI state

Having into account these criteria we end up with a total of 31 APKs ready to be used with our experiments.

We used English as the default language, and for the target languages we used $L = \{$*Spanish, Hindi, Arabian, Russian, Portuguese, French, Italian*$\}$. These set of languages is based on the list of languages with the largest number of speakers. Chinese, Malay, and Bengali were not used in the study — despite being in the list of top languages — because the IBM Translation Service does not provide translations from English to those three languages. With that list of languages, and for each original APK, ITDroid generated an internationalized APK supporting the 7 languages in L, *i.e.,* the analysis was done on 31 original APKs, and 31 automatically internationalized APKs.

B. Open coding

To answer the RQs, we executed ITDroid on the 31 original APKs, which automatically generates internationalized APKs, and then used the statistics and information provided by the ITDroid report. The ITDroid results were validated and analyzed with an open coding inspired procedure. The validation aimed at detecting false positives reported by ITDroid, and identifying the ICCs inducing *i18n* collateral bugs (ICBs) in the apps.

The *i18n* changes were manually analyzed by all the authors with open coding sessions that were supported on a web tool we created. For each ICC assigned to a tagger, the tool showed: (i) the type of change; (ii) the GUI components involved in the change; (iii) a side-to-side comparison of the GUI state (*i.e.,* screnshoot) of the original APK and the same GUI state on a internationalized version, while highlighting the components involved in the ICC; and (iv) a set of input fields for selecting whether the depicted case is a false positive or a *i18n* collateral change, and whether the ICC is a *i18n* bug or not.

The tagging process consisted of two stages. First, the complete set of ICCs where distributed between the five authors by ensuring each ICC was assigned at least to two taggers. This first stage required each tagger to select between 3 main categories to tag (*i.e.,* false positive (FP), ICC or ICB). After this first stage of tagging, we had 273 cases where the taggers agree with all the tags, and 129 conflicts in which there was a disagreement in at least one of the tags.

The second stage, consisted of solving the tagging conflicts. For the conflicts resolution, the cases where distributed between 3 authors. For this process, the web app was modified to show the tags provided by the original taggers; the identities of the original taggers were not disclosed to avoid biasing the conflicts solver.

C. Analysis method

To answer $\mathbf{RQ}_1$, we built a taxonomy of *i18n* changes and bugs, by analyzing the results reported by ITDroid and the open coding process. The ITDroid report includes hard-coded strings, non-internationalized strings, *i18n* changes, and screenshots of the changes. In addition to the taxonomy, we provide in Section V qualitative examples of the *i18n* changes and bugs.

To answer $\mathbf{RQ}_2$ we analyze and report, with representative qualitative examples, the cases marked as *i18n* bugs during the coding phase. In the case of $\mathbf{RQ}_3$, we used the frequencies and statistics collected for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, and report the results grouped by (i) GUI component types involved in *i18n* changes (*e.g.,* button-label, label-image, layout-button), and (ii) language where the *i18n* changes and bugs were detected.

Note that the frequencies in the taxonomy report the *i18n* changes detected by ITDroid in the 7 targeted languages (overall). It is also worth noting that because an *i18n* change is a relationship between two GUI components $GC1$ and $GC2$, there is a dual nature. In the direction of $GC2 \rightarrow GC1$ the relation could be different than in $GC1 \rightarrow GC2$. For example, in the $GC1 \rightarrow GC2$ case, the change could be a miss-alignment, but in the case of $GC2 \rightarrow GC1$ the change could a position-related one. Thus, when reporting frequencies, if the change is the same in both directions we counted it as one instance, otherwise, there are two different *i18n* changes.

All the details of the apps, original and internationalized APKs, the reports generated by ITDroid with the internationalization changes, and the ITDroid code are publicly available within our online appendix [36].

V. Empirical Study Results

In this section we report the answers for each one of the research questions in our study. We present and discuss our findings by using a taxonomy of changes and bugs we built with the results reported by ITDroid and the open coding phase. Fig. 4 presents the taxonomy in which we distinguish *i18n* changes from *i18n* bugs. Concerning the later, there are bugs introduced by developers because the lack of *i18n* practices (*e.g.,* hard-coded strings), and collateral bugs induced by the *i18n* collateral changes. The discussion of the results includes visual examples of the changes and the bugs exhibited by the analyzed apps. In addition, we highlight with ***bold and italic*** the learned lessons and implications for the developers and researchers communities. All values presented in this section do not include the false positive found during the open coding process. In particular, we found 30 false positives out of 402 ICCs reported by ITDroid.

A. RQ_1: What types of i18n collateral changes and bugs are exhibited on GUIs of Android apps?

We found that *i18n* collateral changes and *i18n* code bad practices are more frequent than collateral bugs in the analyzed apps. In the following we describe the results for each category of changes, bad practices, and bugs listed in Fig. 4.

***i18n* Collateral Changes (ICC)**. Based on the layout graphs proposed by Alameer *et al.* [16], [17], ITDroid

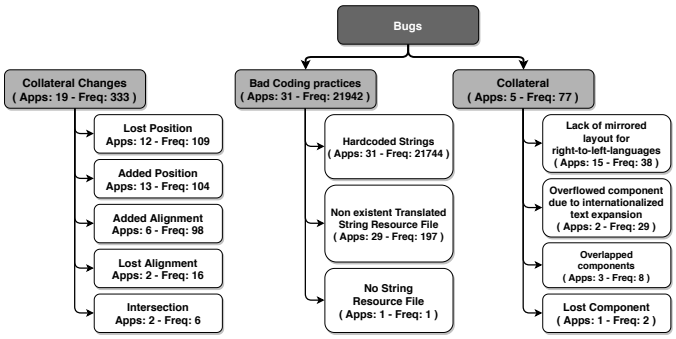


Fig. 4: Taxonomy of *i18n* collateral changes and bugs exhibited on the 31 analyzed Android apps. Each box in the taxonomy reports the number of apps with the change/bug, and the number of times (Freq:) the change/bug was detected.

found instances of 4 types of changes. The most frequent ones are related to components position: *lost position*, *added position* and *intersection*. It is worth noticing that the values of *i18n* changes reported in the the taxonomy refer to individual changes in the apps GUI. Therefore, one action such as shifting an element to other line can have different individual changes, *e.g.,* a case of lost position (*right or left*), a case of added position (*below or above*) and finally, a case of added alignment (*left_aligned or right_aligned*).

Fig. 5 is a representative example of the *lost position* category, in which we can see how the **right** relation between the "close app" and "ask me again" buttons is lost when the internationalized texts in Portuguese expand. We found this type of change in 109 cases (12 apps). Fig. 5 also represents a case of *added position*, because there is a change from **right** to **bottom** between the "close app" and "ask me again" buttons. We found added positions in 104 cases (13 apps).

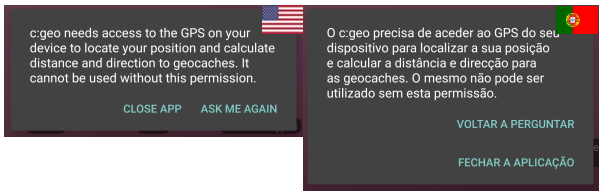


Fig. 5: Example of added and lost relations in the *cge.geocaching* app.

Besides lost and addition of positions, we found 6 cases in which changing to a non-default language generated *intersections* that did not exist in the original APKs. There are two types of intersection: *overlapping* and *breakage of parent-imposed constraints*. *Overlapping* happens because a containment relation is introduced between two components when the app is used in a non-default language. An example of this behavior is shown in Fig. 6: in the English version, the "Select Preset" label and the down-arrow icon are next to each other, however, when language is changed to Arabic, the constraints of the text are not well defined and the text overlaps with the icon.

Fig. 6: Example of an intersection relation in the *com.adermark.forestpondfree* app

The *breakage of parent-imposed constraints* type, groups the cases where a containment relation is lost. Specifically, an intersection relation replaces the lost one, generating a behavior like the one presented in Fig. 7.

Fig. 7: Example of an overlapped component in the *org.ethack.orwall* app

Concerning changes related to components alignment we found 114 instances: 98 cases of *added alignment* and 16 cases of *lost alignment*. Fig. 8 presents an example of *added alignment* between the second and third buttons (from left to right), where after translating to Portuguese, the text content of the third button expands and fills the height of its container. This results in a new bottom-alignment with the second button. However, in the English version the buttons were not aligned.

Fig. 8: Example of an added alignment relation in the *com.android.logcat* app.

Opposite to the previous case, the *lost alignment* category gather all the cases where a relation is removed. For example, Fig. 9 shows a case where most of the buttons lose its top-alignment relation with the first button, when the app is explored in Hindi.

Fig. 9: Example of a lost alignment relation in the *com.android.logcat* app.

***i18n* Bad Coding Practices (IBCP)**. `ITDroid` found 21942 cases of bad coding practices in the 31 analyzed apps. The bad practices are organized in three groups: *hard-code strings*, *non-existing string resources*, and *non-existent internationalization file*. The *hard-coded string* practice refers to strings that are placed directly in the source code. From a internationalization point-of-view, this is a bad practice done by developers, since those strings do not change when the device language is modified. The most common appearance of this behavior is to set a value from Activities or Fragments code into a widget. This bad practice is exhibited 21744 times in the 31 analyzed apps.

The *non-existent string resource* bad practice occurs when an app has no `strings.xml` file. This leads to the extreme

case where all the strings in the application are hard-coded, hindering the creation of an internationalized version of the app. In our study we identified only one app with this behavior: *com.example.android.musicplayer*.

The lack of `strings.xml` file can be also extended to the corresponding files for non-default languages. We called this case as the *non-existent internationalization file*. When considering the 7 target languages used in this study we found that in the analyzed apps, in 29 apps least one of the languages in the list is not internationalized.

The prevalence of these bad coding practices suggests that developers do not use the utilities available at the Android Studio IDE for detecting and extracting hard-coded strings. `ITDroid` is an option for detecting these bad practices, outside of the IDE and without the need of having access to source code. However, if developers prefer tools embedded in the IDE, we encourage them to use the existing features, and customize detection rules for non-existent string resources and non-existing internationalization files (extending the Lint tool available with Android Studio).

i18n* Collateral Bugs (ICB)**. We found 4 types of ICBs induced by *i18n* collateral changes, which group a total of 39 bugs belonging to 5 apps. The most prevalent type of ICB is *overflowed component due to internationalized text expansion* (29 instances). This type of bug is visible in GUIs because it breaks the original design when large texts push the components to be out of their expected dimensions, positions and alignments. For instance, Fig. 5 shows how in the *cgeo.geocaching* app, when changing to Portuguese, the horizontal arrangement of the dialog buttons is pushed to be vertical because of internationalized text expansion in the buttons. When looking into the details of the *cgeo.geocaching* app's layout we found that the bug is exhibited by an Android *AlertDialog*. ***Therefore, developers must be aware that even Android composite widgets are prone to i18n bugs.

We also found that using linear layouts instead of constraints layout is a common error in Android apps. Junior developers prefer linear layouts because are easier to use; constraint layouts are more complex to handle when there is no deep knowledge of the available constraints. ***In addition to being performance friendly, constraint layouts can help developers to avoid issues when dimensions of GUI components are modified dynamically. Therefore, developers should be knowledgeable of the constraint types and be aware that changes in text lengths can break the layout drastically, in particular when changing the default language.***

Other types of ICB, but less frequent, are *lost component*, *lack of mirrored layout for right-to-left-languages*, and *overlapped components*. The former type relates to cases in which a visible component, is pushed out of the display view because a text component is re-dimensioned (see Fig. 10). This does not seem to be a problem at first sight, however, this type of issue could hinder the execution of automated tests that expect certain components to be visible. The *lack of mirrored layout for right-to-left-languages* is a very specific bug that

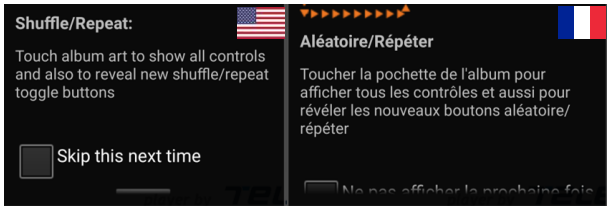


Fig. 10: Example of lost content in *com.teleca.jamendo* app. Due to space limitations, this example contains a snippet of the full screen size example that can be found in our online appendix.

is produced when developers do not consider bidirectionality in their layouts. Bidirectionality means that for languages that read from right-to-left (RTL), UIs should be mirrored to ensure understandability. One example of this bug is Fig. 11. ***To avoid this type of bugs, developers should follow bidirectionality guides that describe how to mirror layouts at the design concept and implementation levels [37], [38]. Static tools can be a solution here, by automatically analyzing and implementing the bidirectionality and RTL guidelines.***

Fig. 11: Example of a lack of mirrored layout for right-to-left–language in the *com.nloko.android.syncmypix* app.

Finally, the *overlapped components* bug is mainly caused by the lack of proper constraints between two elements. This bug can happen between two aligned elements, as it can be seen in Fig. 6, and between an element and its container (see Fig. 12).

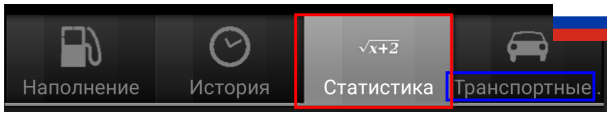


Fig. 12: Example of an overlapping element and its container in *com.evancharlton.mileage* app when internationalized to Russian. The red and blue squares show the overlapping relation

In general, most of the reported ICBs are caused by incorrect definition of constraints on components with text and the lack of tools supporting automated detection and fixing. The bugs induced by internationalized text expansions can also be easily fixed by using the `ellipsize` attribute of the GUI components. In addition, although, `ITDroid` is a partial solution that helps developers to detect IBCPs and ICCs, they have to manually go over the ICCs reported by `ITDroid` to identify ICBs. Therefore, future work could

focus on extending the `ITDroid` approach for automatically detecting the bugs by relying on automated image-based comparisons or by statically detecting (i) constraints incompatibilities and issues, and (ii) missing configurations and resources for enabling bidirectionality.

Other potential impact of the IBCPs and ICBs described here, but not investigated in our study, is related to the behavior of screen readers when IBCPs and ICBs are exhibited in internationalized apps explored by users with visual disabilities. Although it is an aspect not deeply investigated yet, an empirical study by Vendome* et al. *[39] reports that internationalization of assistive content in Android apps is a concern expressed by some developers at Stack Overflow.

B. RQ_2: What i18n collateral changes induce i18n collateral bugs on GUIs of Android apps?

Overflowed component due to internationalized text expansion. This type of bug is generated in most of the cases by position-related changes, since the element that is shifted to a new line lose all its position relations with the elements that were in the same line. An example of this behavior is Fig. 5. Because the button expanded into the next line, the position alignments between the "ask me again" and "close app" buttons are lost, and instead a new **below/down** relation is added.

Lost Component. For this type of bug, the 2 cases we found are related to internationalized text expansions that push other components out of the user view. In the layout graphs, this is represented as a lost position of the components pushed out of the view, since those components (after the change) are not visible anymore on the UI. The example in Fig. 10, shows how there is a button at the end of the English version, but, the button is not visible in the French version. This is represented in the LayoutGraph as an element that does not appear in the graph of internationalized version.

Lack of mirrored layout for right-to-left-languages. This type of bug is mainly represented as the lack of of alignment and position changes, since layouts for RTL languages should mirror the position of most of its elements. For example, all **right** alignment and position relations should became **left** alignment and position relations. Note that we found only 2 cases with `ITDroid` during the open coding of the ICCs. However, such a small number of cases is explained because the layout graph-based approach [16], [17] used by `ITDroid` focuses on detecting changes between layout graphs. The *lack of mirrored layout for RTL languages* is in fact a bug that should be identified when no changes are detected (*i.e.,* there is no mirroring) in components such as [37]. Therefore, we manually analyzed the screenshots collected by `ITDroid` when the emulator was configured for Arabic. We found 15 apps suffer from this bug (see Fig. 11). ***In order to detect this type of bug, for the specific case of RTL languages, the layout graph-inspired detection should look for components not changing positions.***

Overlapped components are exhibited mainly by intersection and lost alignment relations. This bug considers all the cases where one element overflows its container, for instance, the existing alignments of right and left borders between contained elements and its container are broken.

C. RQ_3: What are the GUI components and languages more prone to i18n collateral changes and bugs?

Fig. 13 presents the amount of ICC and ICB distributed along the languages used with our study. Arabic is the language with more ICCs accounting for a total of 151 collateral changes, nonetheless, 39 ICCs were considered as ICBs during the manual tagging phase. Most of the ICCs generated when internationalizing the analyzed apps to Arabic, are explained because the lack of awareness of the RTL nature of language, when designing the layouts. As already mentioned in the previous section, ***layouts for Arabic versions of apps should adapt alignment features to achieve bidirectionality; nevertheless this might generate a conflict for developers since it might require to create two different versions of the app, however, since API 17, Android apps accept RTL orientation by defining this behavior in the layout files (see [38]). For example, developers might want to use the `paddingStart` attribute instead of `paddingLeft`, which delegates the orientation to the operating system [38].***

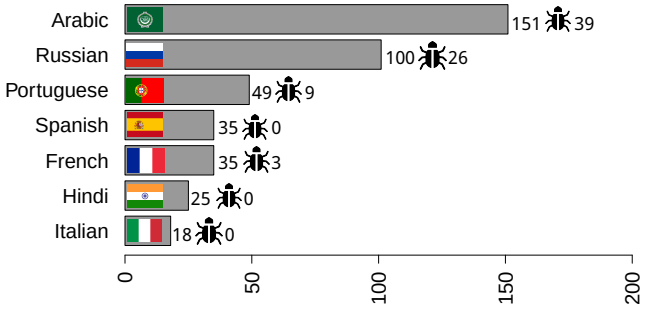


Fig. 13: Distribution of lCC and ICB in the analyzed languages. The values next to the bar are the amount of ICCs, while the amount shown next to the bug icon is the amount of ICBs.

Russian is the top-2 language involved in ICCs, with 101 cases reported by `ITDroid`. The ICCs generate 26 collateral bugs distributed in *Overflowed component due to Internationalized text expansion* and *Overlapped components* (See Fig. 12) As in most of the cases reported in this study, ***the bugs are induced by text expansions that are not properly controlled with layout constraints or by shortening texts using an ellipsis* (i.e., *using the `ellipsize` attribute [40]). Using English as the default language in Android apps, and not conducting proper i18n testing, makes apps prone to i18n collateral changes and bugs because English is among the languages with smallest words [41].***

Fig. 14 reports the distribution of ICC and ICB at the GUI component level. `TextView` is the top component, with 125 ICCs but only 6 ICBs. A similar behavior can be found with `LinearLayout`, where there are 79 ICCs but only 5 were

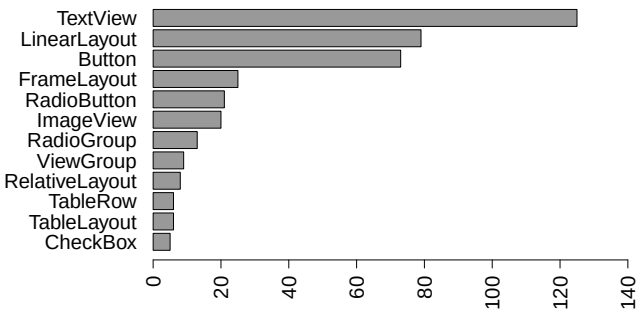


Fig. 14: GUI components involved in the ICCs detected by `ITDroid`.

tagged as ICBs. In contrast, `Button` the top-3 component in terms of ICCs, is also the one with most ICFs: 22 of the 73 ICCs where tagged as ICBs. Note that ICCs related to *Buttons* are commonly found in custom `Dialogs` where layout constraints are not properly defined.

D. `ITDroid` limitations

`ITDroid` relies on a ripper for automatically exploring the UI (Section III-C). In particular, our implementation follows a DFS strategy for exploring the app, which has been demonstrated to have code coverage limitations [9]. Fig. 15 depicts the distribution of method coverage achieved by the `ITDroid` ripper with the 31 analyzed apps. On average, `ITDroid` automatically executed 23% of source code methods in the apps. Another limitation in `ITDroid` is that it only translates the strings that are defined in resources files, *i.e.,* hard-code strings were not automatically translated by the current implementation of `ITDroid`. Therefore, it is worth noticing that the results reported in our study should be considered as a lower bound of the *i18n* collateral changes (ICC) and bugs (ICB). Future work will be focused on improving the `ITDroid` code coverage and including hard-coded strings in the translation process. Improving the ripping strategy, adding a random exploration mode, or even using a tool like Sapienz [12] are avenues for future work that should be investigated to improve code coverage achieved by `ITDroid`.

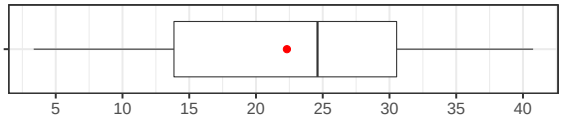


Fig. 15: Method coverage achieved by `ITDroid` during the automated exploration of the analyzed apps.

VI. Threats to Validity

External validity. This type of threats are concerned to whether the results can be generalized to other settings or populations. In this study we used a sample of 31 Android native open source apps, which is not representative of the whole population of Android apps; therefore, we can not generalize our results to the whole set of Android apps that include hybrid and mobile web apps. Additionally, it is known that some popular apps have mechanisms to avoid being repackaged, this behavior might change the result of the tool execution. Future work should be devoted to replicate the study on a larger sample of apps; also, similar approaches should be implemented for Android hybrid and mobile web apps and other platforms such as iOS. In addition, although we analyzed ICCs and ICBs for 7 languages from the top-spoken languages list, we can not claim that our results generalize to other languages with different characteristics, *e.g.,* isolating (Chinese) or agglutinating (German) languages.

Internal validity. Concern confounding factors in the independent variables that can affect the results (*i.e.,* dependent variables). The apps sample is a potential threat, because they could belong to a specific category. However, we reduced this threat by random sampling the apps from a publicly available repository (F-Droid) that has been widely used by other researchers; our sample is diverse in terms of size and categories (see online appendix). Another potential threat is the choice of target language as the independent variable for the number of ICCs and ICBs, however, in our study we executed the experiment on a set of 7 different languages to avoid any bias introduced by the language choice. Finally, the choice of a ripper that follows a DFS strategy is a potential threat to internal validity, because despite of trying to explore as many states as possible, rippers are known to have limitations. Therefore, we recognize that it is possible that there are more GUI states in the apps that were not reached by the ripper. In that sense, our results are reported as a lower bound of the number of ICCs and ICBs in the analyzed apps.

VII. Conclusions & Future Work

Although `ITDroid` was not designed with the purpose of automatically internationalizing apps, since the translation process is implemented with a strategy pattern, any user could implement its own `ITDroid` that works with another translation service. In our case we used the IBM Watson engine, but any translation engine can be easily plugged.

Despite the existence of mechanisms for supporting mobile apps internationalization, issues such as collateral changes, bad practices and collateral bugs were found in the analyzed apps. Therefore, existing practices should be promoted more in the practitioners community and researchers should envision and implement tools for automated detection of *i18n* bugs in the case of issues induced by bidirectionality and text expansions. In addition, existing approaches such as the usage of graph-layouts must be revised to consider issues related to RTL languages.

Concerning `ITDroid` capabilities, future work will be dedicated to improving the language translation coverage. Additionally, implementing mechanisms for automatically identifying *i18n* bugs is an interesting avenue for research. Finally, the automated app exploration could be improved by combining different automated testing strategies, *e.g.,* monkey testing, model-based testing and ripping.

REFERENCES

[1] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 477–487. [Online]. Available: https://doi.org/10.1145/2491411.2491428

[2] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[3] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 291–300.

[4] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 353–363.

[5] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings in web applications," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882306

[6] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 516–536, April 2013.

[7] X. Wang, C. Chen, and Z. Xing, "Domain-specific machine translation with recurrent neural network for software localization," *Empirical Software Engineering*, Apr 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09702-z

[8] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 613–622.

[9] S. R. Choudhary, A. Gorla, and A. 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)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[10] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[11] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[12] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[13] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1013–1024. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568229

[14] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, June 2017.

[15] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, Sep. 2015.

[16] A. Alameer, S. Mahajan, and W. G. Halfond, "Detecting and localizing internationalization presentation failures in web applications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 202–212.

[17] A. Alameer and W. G. Halfond, "An empirical study of internationalization failures in the web," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 88–98.

[18] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 215–226.

[19] V. Dagiene and R. Laucius, "Internationalization of open source software: framework and some issues," in *ITRE 2004. 2nd International Conference Information Technology: Research and Education*, June 2004, pp. 204–207.

[20] J. M. Hogan, C. Ho-Stuart, and B. Pham, "Key challenges in software internationalisation," in *Proceedings of the Second Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32*, ser. ACSW Frontiers '04. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 187–194. [Online]. Available: http://dl.acm.org/citation.cfm?id=976440.976469

[21] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 32–41. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250739

[22] X. Xia, D. Lo, F. Zhu, X. Wang, and B. Zhou, "Software internationalization and localization: An industrial experience," in *2013 18th International Conference on Engineering of Complex Computer Systems*, July 2013, pp. 222–231.

[23] S. Mahajan and W. G. J. Halfond, "Detection and localization of html presentation failures using computer vision-based techniques," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[24] Apple, "Testing your internationalized app. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPInternational/TestingYourInternationalApp/TestingYourInternationalApp.html."

[25] Y. Arnatovich, H. B. K. Tan, S. Ding, K. Liu, and L. K. Shar, "Empirical Comparison of Intermediate Representations for Android Applications," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2014, pp. 205–210.

[26] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12 382–12 394, 2018.

[27] "Apktool. https://code.google.com/p/android-apktool/."

[28] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[29] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[30] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 33–44.

[31] S. Liñán, L. Bello-Jiménez, M. Arévalo, and M. Linares-Vásquez, "Automated extraction of augmented models for android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 549–553.

[32] K. Jamrozik and A. Zeller, "Droidmate: A robust and extensible test generator for android," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2016, pp. 293–294.

[33] Y. Cao, G. Wu, W. Chen, and J. Wei, "Crawldroid: Effective model-based gui testing of android apps," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, ser. Internetware '18. New

York, NY, USA: ACM, 2018, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/3275219.3275238

[34] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 233–244. [Online]. Available: https://doi.org/10.1145/3106237.3106275

[35] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. L. Vásquez, G. Bavota, C. Vendome, M. D. Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," *CoRR*, vol. abs/1802.04749, 2018. [Online]. Available: http://arxiv.org/abs/1802.04749

[36] C. Escobar-Velásquez, M. Osorio-Riaño, J. Dominguez-Osorio, M. Arevalo, and M. Linares-Vásquez, "Itdroid: Internationalization of android apps. https://thesoftwaredesignlab.github.io/ITDroid/."

[37] Google. Bidirectionality. https://bit.ly/3c61pyd.

[38] ——. Support layout mirroring. https://bit.ly/2yDkx8T.

[39] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can everyone use my app? an empirical study on accessibility in android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 41–52.

[40] Google. android:ellipsize. https://bit.ly/2XEmwSN.

[41] R. Smith, "Distinct word length frequencies: distributions and symbol entropies," *Glottometrics*, vol. 23, pp. 7–22, 2012.