Detecting Behavior Anomalies in Graphical User Interfaces

Vitalii Avdiienko♣, Konstantin Kuznetsov♣, Isabelle Rommelfanger♣, Andreas Rau♣ Alessandra Gorla♠, Andreas Zeller♣
♣*Saarland University* ♠*IMDEA Software Institute*
Saarbrücken, Germany *Madrid, Spain*

***Abstract*—When interacting with user interfaces, do users always get what they expect? For each user interface element in thousands of Android apps, we extracted the Android APIs they invoke as well as the text shown on their screen. This association allows us to detect *outliers:* User interface elements whose text, context or icon suggests one action, but which actually are tied to other actions. In our evaluation of tens of thousands of UI elements, our BACKSTAGE prototype discovered misleading random UI elements with an accuracy of 73%.**

One of the central principles in user interface design is *the principle of least astonishment*—that is, a user interface element should behave in a manner consistent with how its users expect it to behave. Such user expectations typically stem from *similar programs* which users are familiar with: If a UI element says "Print", "Save", "OK", "Close", or "Cancel", our experience with other programs using these labels gives us an idea of what to expect; and if the result does not match our expectation, we see this as a problem.

The possibly most dangerous mismatches, however, are those we do not even notice. Figure 1 shows the signup screen of TRIPWOLF, a popular travel guide app from the Google Play Store. Signing up to the service is done by entering an e-mail address and clicking on "Join TRIPWOLF". The interesting thing about this Sign-up button is that it not only sends the e-mail address, but also the *precise user location* to the TRIPWOLF servers, using the *LocationManager* API—this button shows a mismatch between user expectation and actual behavior.

In this paper, we check the *advertised* functionality of UI elements against their *implemented* functionality: "This Signup button should not send a location, but it does". The idea is to mine *app stores* – containing hundreds of thousands of apps with graphical user interfaces – and model users' expectations with respect to a particular UI element. The *expected behavior* results from similar UI elements in other apps.

The approach identifies *arbitrary mismatches between what a UI element shows and what it actually does.* Simple UI programming mistakes are also detected: If a programmer included a button named "Signup" that always stays on the same screen, or sends a text message, or prints a file, this error could also be caught. Even translation issues stemming from automatic app conversion mistakes can be recognized.

In contrast to other work leveraging app collections to detect general mismatches between advertised and implemented behavior [2], [5], [6], BACKSTAGE detects particular anomalies at the UI level rather than the app level. Moreover, it is not restricted towards stealthy behavior or privacy issues unlike existing work checking for mismatches at the GUI level [3]. Instead, it provides a general means to detect mismatches between advertised and implemented behavior at the GUI level.

Fig. 1. For each UI element, BACKSTAGE determines the APIs it triggers, and checks for anomalies: The TRIPWOLF Signup button sends the current location to TRIPWOLF servers.

Our approach consists of five steps, summarized in Figure 2:

App Collection We start with a collection of thousands of ANDROID apps, all taken from the Google Play Store by using the ANDROZOO [1] dataset.

Mining UI Elements. We statically analyze the code and resources of each app to identify its set of UI elements, including those that would be set or changed dynamically.

Context and APIs. For each UI element, we extract its *associated text*—both *labels* shown on the element itself, *context* from the surrounding screen, as well as the APIs that would be triggered by the element. In our example, "Join TRIPWOLF" uses *LocationManager.getLastKnownLocation()* to retrieve the precise current location, and *startActivity* to switch to the next input screen.

Cluster Analysis. From associated text of all UI elements, we cluster their verbs and nouns into 250 *concepts*—clusters of words with a minimal *semantical distance* using a WORD2VEC model [4]. For each UI element, we determine the distance between its text and the concepts. A button named "Share", would be semantically close to the concepts "friend" (to share) and "finances" (a share).

For each button, we also extract the typically used APIs. The "normal" behavior of a Sign-up function is to access the network via *android.net* or *org.apache.http*. Several sign-

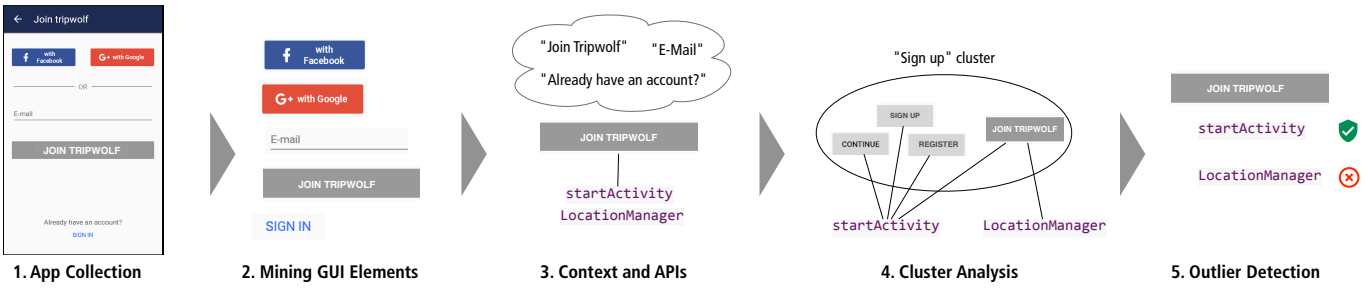

Fig. 2. **Process Pipeline** - For each app in a collection (Step 1), the app code and UI descriptions are statically analyzed to extract UI elements (Step 2). For each UI element, the textual description and context on the screen are extracted, as well as the APIs that it triggers (Step 3). Across all apps, UI elements with semantically related descriptions are clustered (Step 4) and an outliers analysis is performed (Step 5)—the Sign-up button from Figure 1 that sends out the current location is an anomaly.

TABLE I
ACCURACY FOR DETECTING DISCREPANCIES BETWEEN *advertised* AND *implemented* FUNCTIONALITY BY ANALYZING SYNTHETIC UI ERRORS (MUTATIONS).

	Precision	Recall	Accuracy	Specificity
Random Mutants^A	75%	67%	73%	79%
High Distance Mutants^B	76%	71%	75%	79%
Crossover Label^C	69%	48%	65%	80%

up functions also access *android.telephony* to access the country code of either the current network or the inserted SIM card. The *android.location* package also is frequently used—but only to access the current local time.

Outlier Detection. For each concept, we use outlier detection to identify those UI elements that invoke uncommon APIs, indicating differing (and possibly unexpected) behavior. Accessing the current precise location is rarely used in the Signup cluster—and thus, the TRIPWOLF Signup button is flagged as an anomaly.

RESULTS AND FUTURE WORK

Since our analysis uses most widely used apps, with a high level of maturity, visible UI errors typically are already detected, reported, and fixed. We resort to a well-established scheme used to evaluate testing techniques. By injecting *synthetic UI errors* (mutations), we measure how effective the presented technique can detect *general* UI mismatches. Specifically, we take existing buttons and change their labels such that they would no longer match the APIs used; and then evaluate whether these mutations are detected as anomalies.

Table I shows the results. Having random text replacements as a baseline (A), the algorithm detects 67% of UI errors. Obviously, the semantic distance between the given and the correct label is crucial. If the distance is high, as indicated by our "high distance" mutations (B), they are identified. But if the distance is low, i.e. the items are semantically close, an API mismatch would not be detected. The labels "Stop", "Abort", or "Cancel", for instance, are semantically almost equivalent and the presented technique will have a hard time differentiating them, just as humans will. But even small programming issues, e.g. the programmer confuses two button labels (C), can be automatically detected in every second case. All these results should be interpreted from the standpoint that to the best of our knowledge, there is no other approach which would detect such mismatches.

This work is the first to generally check the *advertised* functionality of UI elements against their *implemented* functionality. To this end, thousands of existing UI elements are analyzed for text and context shown to the user, and clustered by common concepts. In each cluster, *outliers* are detected—that is, UI elements that use different APIs than the others. This approach is general and effective.

Despite these advances, we see the work not as an end—but rather as a beginning of a new field "UI mining", where we would mine thousands of UIs to learn common vs. uncommon features in behavior, appearance, and process. Our future work will focus on the following topics:

Dynamic behavior. Despite the ease of static analysis, we are considering using additional dynamic analysis and exploration to assess dynamic features. Most notably, we want to *validate* reported anomalies by creating test cases that demonstrate the actual API access.

Automatic repair. Detecting that an UI element label does not match its behavior allows for automatic suggestions of better labels. One idea we are investigating is to identify labels of UI elements that use similar APIs and suggest them as automatic repairs: "This button should be named 'Send'."

Alternate domains and UI features. Besides ANDROID apps, there are several other domains with programs whose UIs could be mined, such as desktop applications. Besides looking for anomalies between text and behavior, one might also examine anomalies in visual presentation ("The 'Send' button should be highlighted"), layout, process, or visual images—opening the door to general automatic anomaly detection and recommendations for UI design.

To facilitate assessment, reproduction, and extension, our prototype BACKSTAGE is publicly available—from source code to build instructions to evaluation scripts. For details, check out the project site at

http://www.st.cs.uni-saarland.de/appmining/backstage/

REFERENCES

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[2] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

[3] J. Huang, X. Zhang, L. Tan, P. Wang, and B. 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.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[5] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.

[6] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1354–1365, New York, NY, USA, 2014. ACM.