When App Stores Listen to the Crowd to Fight Bugs in the Wild

Maria Gomez, Matias Martinez, Martin Monperrus, Romain Rouvoy

When App Stores Listen to the Crowd to Fight Bugs in the Wild

María Gómez, Matias Martinez, Martin Monperrus, Romain Rouvoy
Inria Lille - Nord Europe
University of Lille 1, France
Email: firstname.lastname@inria.fr

***Abstract*—App stores are digital distribution platforms that put available apps that run on mobile devices. Current stores are software repositories that deliver apps upon user requests. However, when an app has a bug, the store continues delivering defective apps until the developer uploads a fixed version, thus impacting on the reputation of both store and app developer. In this paper, we envision a new generation of app stores that: (a) reduce human intervention to maintain mobile apps; and (b) enhance store services with smart and autonomous functionalities to automatically increase the quality of the delivered apps. We sketch a prototype of our envisioned app store and we discuss the functionalities that current stores can enhance by incorporating automatic software repair techniques.**

I. Introduction

Increasingly, mobile devices are infiltrating most of our daily activities, and consumers are using more and more mobile applications (apps for short). They download apps from dedicated app stores (Google Play, Apple Store, Amazon Appstore, etc.), which are in constant growth. In 2013, Google Play Store reached over 50 billion app downloads [1].

Previous studies have shown that these stores deliver a significant portion of buggy apps to mobile devices. For instance, we have identified $10,658$ buggy-suspicious apps in a dataset of $46,644$ apps collected from Google Play Store [5]. Currently, the repair activity of defective apps is manually performed by app developers. After fixing the identified bugs, developers upload the new version (with the fixes) to the app store. Unfortunately, the time between the release of the defective app and the fixed version can be long. As a consequence, in the meantime, consumers continue to download defective apps and to experience undesired behaviors. This paper proposes the idea of an app store that orchestrates hot patches in production to overcome this issue.

To fight bugs in the wild, we devise the following approach. Defective apps are detected by constant analysis of user feedback in the form of reviews and ratings. While previous research studied user feedback in app stores [8], [6], [9], [4], none of them proposed mechanisms to exploit feedback by app stores themselves, autonomously. Next, the app store generates tentative hot patches for fixing app crashes. Finally, it monitors the performance of the fixed apps in the wild to learn about the correctness of the repairing and patch delivery process. The proposed strategy allows the store to detect, repair and validate defective apps without developers' intervention.

[1] https://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down

This orchestration is a feedback loop, since the app store itself takes decisions based on the outcome of the previous ones. If a patch generation technique fails (*e.g.*, the fixed app still crashes), the store learns from these failures; If a delivery strategy distributes patches to the wrong set of devices (*e.g.*, not all the devices suffer from the same bug), the store detects it. The app store continuously monitors both the fixed apps' execution and the user's feedbacks as an oracle of the autonomous improvement process. To sum up, we envision app stores that go beyond the role of app repositories and become intelligent agents. In this vision, app stores autonomously take decisions to improve the quality of experience of their customers.

To sum up, our contributions are:

- A blueprint of a smart app store capable of automatically detecting, generating, and validating patches using crowd-sourced information;
- A prototype implementation of such a smart app store;
- A preliminary evaluation over one real defective application from the Google Play Store.

The reminder of the paper is organized as follows. Section II describes the key components that form the infrastructure. Section III presents a prototype to demonstrate the feasibility of our proposal. Section IV reviews the related work. Section V concludes the paper.

II. Vision of a Smart App Store

In our vision, app stores will incorporate two types of mechanisms in order to fight bugs in the wild: 1) *proactive mechanisms*, the store anticipates the emergence of bugs wild upon submission of new apps using static app checkers [5]; and 2) *reactive mechanisms*, the store acts when users experience bugs after installing apps. While this paper focuses on the latter, both mechanisms are complementary and can be activated simultaneously.

The smart app store we envision is capable of: a) retrieving and consolidating information from app's users and from devices running those apps, b) identifying apps that work unexpectedly in the wild (*i.e.*, contain bugs), c) generating patches automatically for the identified defective apps, d) delivering fixed apps contextually, and e) validating the generated patches automatically from information received from devices.

Figure 1 depicts an overview of the smart app store we propose. This infrastructure builds on the principles of autonomic computing [7] and presents an architecture based on a feedback loop model, which combines five components:

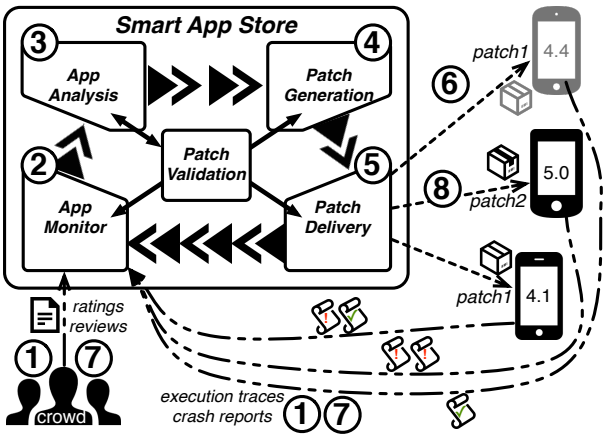


Fig. 1. Overview of the *Smart App Store* vision.

App Monitor, *App Analysis*, *Patch Generation*, *Patch Delivery*, and *Patch Validation*.

The functionality of a smart app store is defined as follows. The app store makes available apps to users. Once an app is installed and executed on mobile devices, the App Monitor component receives information sent from devices and from user feedbacks (reviews, scores, bug reports, etc.) (step 1, 2). This information is associated with the version number of the app. Then, the App Analysis component analyzes the retrieved information to decide if an app exercises a wrong behavior—*i.e.*, it has a bug (step 3). In that case, the Patch generation component tries to synthesize candidate repairs using automatic repair approaches, such as GENPROG [11] (step 4). For each candidate patch, the component defines a new release of the app that includes the patch (step 5). Then, when the store receives a download request for a defective app, the Patch delivery component delivers—when possible—an alternative (patched) version of the app (step 6).

Afterwards, the store keeps monitoring these patched apps to assess the effectiveness of the corresponding patches (step 7). For that, it uses the collected crowd-sourced information from devices and from users' feedback that run those apps. Finally, when the store receives a new download request, it delivers–if exists–a release that does not exercise the bug and has better users' feedback (step 8). The remainder of this section details the role and relevance of each component.

A. Monitoring Experience from the Crowd

The App monitor component receives and stores information coming from the crowd. We identify three types of information: *1) comments and rankings* that app users publish in the store; *2) device's features—e.g.*, the operating system version that runs the app; *3) apps' execution context—e.g.*, exception traces. The App monitor component associates the retrieved information with the corresponding app version. For example, the store keeps track of the device models and OS versions that run a given app in the crowd.

B. Analyzing App's Quality of Experience

The App analysis component analyzes the collected artifacts to detect apps that exercise an incorrect behavior—*i.e.*, contain a bug. To detect bugs, the component is able to process the different sources of information we identified previously.

1) Analyzing User Feedbacks: The App analysis component incorporates functionality to continuously supervise the user feedback published in the store (*e.g.*, reviews, ranks). This component can gather user feedbacks from different sources to alert the presence of bugs. A listener continuously mines user reviews in order to identify apps that accumulate error-related reviews—*i.e.*, those which discuss mainly about buggy behavior. The App analysis component therefore computes trends in the ratio of buggy reviews and other reviews over time. When an app reaches a predefined threshold number (n) of error-related reviews, the App analysis component flags the app as *buggy-suspicious* and the store starts the inspection of the suspicious app. Detailed information about the techniques that this component leverages are available in [5].

2) Analyzing App Traces: The App analysis component analyzes execution traces to confirm that a suspicious app exhibits failures. There are many causes that induce app failures—*e.g.*, connectivity problems, memory exhaustion, or insufficient permissions. If the failures are handled inadequately in the app code, then the app throws an unhandled exception and the operating system terminates the app.

3) Controlling the Monitoring of Apps: In this paper, we propose a two-step heuristic to detect defective apps. First, a *static* analysis algorithm inspects the user feedback to know if users experience bugs. Then, in order to confirm that a suspicious app has a bug, the store monitors execution traces to detect the rise of exceptions.

C. Planning Alternative Releases for an App

The Patch generation component receives as input a defective app (from the App analysis component) and finds candidate patches that eventually solve the bug. To synthesize candidate patches, the component uses existing automatic software repair approaches, such as GENPROG, which is capable of repairing binary [10] and source [11] code. Current app stores only have access to the bytecode of apps. Nevertheless, future app stores could also enable developers to upload the source code of their apps to allow for more powerful repairing techniques.

To repair a bug, the Patch generation component can produce several candidate patches. The rational behind generating multiple candidate patches is: a) the existence of several solutions to fix a bug; b) the approach has a weak correctness bug oracle, then producing incorrect candidate variants (no solution). The component generates a new version of the defective app for each candidate patch that it synthesizes. We denote the set of patched versions as *variants*. Initially, patches are assigned the *under-validation* state. After deployment and checking their correction, the state changes to *validated*.

D. Executing the Delivery of Apps

When the store receives a download request for an app, which has been previously identified as defective, the Patch delivery component checks if there exists in the store at least one patched version of the requested app. The delivery contemplates two cases in priority order: *1)* delivering *validated* variants; *2)* delivering *under-validation* variants—*i.e.*,

the patches are in an evaluation process to assess their validity. The component delivers a validated variant, if present. Otherwise, it delivers an under-validation variant. In the absence of variants, the store delivers the original app.

To select the variant for delivery, the component analyzes and compares the available information regarding all the variants. The store implements different app selection heuristics. For example, one heuristic delivers the variant that provides better performance (*i.e.*, less crashes) observed in similar devices (*i.e.*, same OS version). Another heuristic is based on user feedback: it selects the variant with the highest user ranking.

After delivery, the store registers the variant that delivers to each user. The pairs *user-variant* allows the store to monitor the performance of the patched apps in the crowd for, later, deciding whether the applied patches are effective or not.

Using information from the crowd, the App monitor component validates the generated patches and removes those *under-validation* variants that continue exhibiting wrong behavior (not available for subsequent delivery). The component applies similar heuristics to those ones previously presented, for example, discarding variants whose rankings are worse than those from the original app.

The component passes a variant from *under-validation* to *validated* after, for instance, delivering and observing a correct behavior for a given number of users. In this case, the Patch validation component notifies all the users that run a different *variant* of the app that a new fixed version is released and is available for downloading.

III. Preliminary Results

We have implemented a prototype of the smart app store to demonstrate the feasibility of our proposal. To illustrate this approach, we use an Android app that contains bugs: the *PocketTool*[2] app downloaded from Google Play Store. The subject app enables the personalization (*i.e.*, download/install textures and skins) of the popular game *Minecraft*[3]. Once the app is launched, if the user clicks on the button "Level Editor" then the app crashes. Figure 2 (a) shows the screenshot of the crash thrown by the *PocketTool* app.

A. Implementation of the Smart App Store Components

Monitoring and Analyzing User Feedbacks. The healing process starts when the store observes user reviews complaining about bugs and crashes. To identify user reviews that treat themes related to bugs, we extract topics discussed in the corpus of reviews using *Topic Modelling*. Our system classifies as *error-related* reviews the ones which are mainly composed by topics related to bugs and failures. Then, it flags as *buggy-suspicious* the apps whose ratio of error-related reviews reaches a predefined threshold (cf. [5] for implementation details). Our system enables the identification of $10,658$ buggy-suspicious apps in our dataset (composed of $46,644$ apps) collected from Google Play Store. This system enabled the identification of the subject app used in this experiment.

Monitoring and Analyzing Execution Traces. This step is triggered when an app is flagged as buggy-suspicious. Then, the App monitor component uses the Android Logging system[4] (`logcat`) to monitor exceptions raised by the buggy-suspicious app. In Android, the system collects debug information from apps and from the system in logs, which can be viewed and filtered by the logcat. The logs include stack traces when an app throws an error. We have implemented a listener that monitors the logcat and subscribes to error logs. The listener notifies to the store whenever an exception is detected in an app. Figure 2 (b) shows the exception trace (extracted from the logcat), which is thrown by the *PocketTool* app. We observe that the exception is related with a `NullPointerException`. The exception trace reveals some methods implemented in the *PocketTool* app (lines 5–6 in bold), and other methods defined internally by Android libraries (lines 1–3).

Planning Alternative App Releases. First, the Patch generation component processes the exception trace and extracts the n frames that refer to the suspicious app. In our example, the 2 frames that start with the package name of the app (`com.snowbound.pockettool.free`). From each suspicious frame, it extracts the name of the suspicious method and the class that implements it. In our example there are 2 suspicious methods (cf. Fig. 2, lines 5 and 6): `getWorldList` and `onCreate`, defined in the class `LevelSelector`.

Next, the component creates n different patches, where n is the number of suspicious methods. In our implementation, the patch wraps the code defined inside the suspicious methods with a `try/catch` block to capture the runtime exceptions that are not handled by the methods. We create 2 patched versions of the defective app, where each patch wraps a different suspicious method. To inject the patches, we have implemented a Java program[5], which instruments the bytecode of Android apps using Dexpler [2].

Executing the Deployment of Apps. Finally, the different patched app versions are deployed in different user devices. First, the buggy app is uninstalled, and then the new patched version is installed. To communicate with devices, our implementation relies on the *Android Debug Bridge* (`adb`)[6]. `adb` is a command line tool (included in the Android SDK), which acts as a middleman between a host and an Android device. We use `adb` to remotely install/uninstall apps and to read the logcat.

To validate the candidate patches, the App monitor component observes if the fixed app throws the same exception as the original app when running on devices. If the exception still arises, then the patch is considered as invalid and the store discards it. On the contrary, if the exception disappears after patching, then the patch is considered as valid and the store will deliver it in subsequent app requests.

B. Experimental Results

We observe that the patch applied in the `onCreate` method (Patch2) avoids the crash, whereas the patch applied

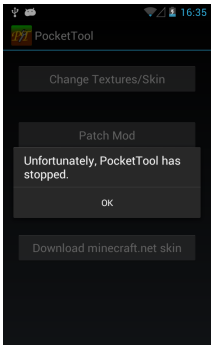
[2]https://play.google.com/store/apps/details?id=com.snowbound.pockettool.free

[3]https://play.google.com/store/apps/details?id=com.mojang.minecraftpe

[4]http://developer.android.com/tools/help/logcat.html

[5]Our instrumentation program is available for download: https://www.dropbox.com/sh/u3ffy1lw85opww8/AACBLu2zcTCUNgXAFh7dpDbma

[6]http://developer.android.com/tools/help/adb.html



(a) Screenshot

```
FATAL EXCEPTION: main
java.lang.RuntimeException: Unable to start activity ComponentInfo
{com.snowbound.pockettool.free/com.snowbound.pockettool.free.LevelSelector}:
java.lang.NullPointerException
1:  at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2059)
    ...
2:  at com.android.internal.os.ZygoteInit.main(Native Method)
3:  at dalvik.system.NativeStart.main(Native Method)
4:  Caused by: java.lang.NullPointerException
5:  at com.snowbound.pockettool.free.LevelSelector.getWorldList(LevelSelector.java:78)
6:  at com.snowbound.pockettool.free.LevelSelector.onCreate(LevelSelector.java:43)
7:  at android.app.Activity.performCreate(Activity.java:5008)
    ...
```

(b) Exception trace

Fig. 2. (a) Screenshot of the subject app crash. (b) Exception trace raised by the subject app (in bold the methods defined in the app).

in the `getWorldList` method (Patch1) continues throwing the exception. Therefore, the store learns that Patch1 is not effective and automatically discards it.

Our proposed implementation takes as input *.apk* files and works remotely with devices without requiring USB connection. We run the experiments in a rooted device Google Nexus S with Android 4.1.2. We generated 2 patched versions of the *PocketTool* app (original size 395 Kb). Each patch rewrites the bytecode of the subject app to inject a `try/catch` block that wraps the code defined in the methods `getWorldList` (Patch1) and `onCreate` (Patch2), respectively. The size of the two patched apps is 439 Kb, and the total time to rewrite the bytecode and redeploy the apps in the device is 51 seconds, which we consider is an acceptable overhead.

IV. Related Work

Recent approaches have investigated different sources of information available on app stores: AR-Miner [8], Wiscom [4], Harman et al. [6], Pagano and Maalej [9]. The aforementioned approaches provide significant analysis about the user feedbacks available on app stores. Nevertheless, none of them provide mechanisms to exploit the user feedback by the stores themselves.

Azim et al. [1] present a self-healing approach to automatically detect failures and patch Android apps to avoid crashes. As in our approach their patching strategy rewrites the bytecode of apps to insert `try/catch` blocks to wrap methods that throw unhandled exceptions. Nevertheless, our approach generates different patches for a buggy app and uses user feedbacks to evaluate the feasibility of the patches. Franz et al. [3] define an app store which generates different versions of an app, functionally identical, upon reception of download requests. The goal of their work is to reduce the vulnerability of apps. In our approach, we generate and distribute app versions with differences in functionality. These differences consist of candidate patches to validate. In contrast to previous work, our goal is to engineer smart app stores that exploit user feedbacks and incorporate repairing techniques to avoid the distribution of defective apps among users. To the best of our knowledge, we propose the first autonomic computing approach to monitor and repair mobile app crashes in the wild that continuously improves its autonomous strategies based on crowd feedback.

V. Conclusion

In this paper, we have presented the vision of app stores that exploit user feedbacks and take autonomous decisions to improve themselves. A smart app store is able to automatically detect defective apps and provide different patches to avoid bugs. After deploying patches, the store learns which the effective patches are and improves its repairing strategy. We have presented a prototype implementation and its usage on a real defective app from Google Play Store.

References

[1] M. T. Azim, I. Neamtiu, and L. M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 623–628, New York, NY, USA, 2014. ACM.

[2] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[3] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[4] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1276–1284, New York, NY, USA, 2013. ACM.

[5] M. Gomez, R. Rouvoy, M. Monperrus, and L. Seinturier. A Recommender System of Buggy App Checkers for App Store Moderators. Research Report 8626, Inria Lille, Oct. 2014.

[6] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: MSR for app stores. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 108–111, 2012.

[7] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[8] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. AR-miner: Mining informative reviews for developers from mobile app marketplace. 2014.

[9] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 125–134, July 2013.

[10] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[11] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.