Indian J.Sci.Res. 17(2): 492-502, 2018 ISSN: 0976-2876 (Print)
ISSN: 2250-0138(Online)

PROSPECTIVE RESEARCH ASPECTS IN SOFTWARE ENGINEERING FOR ANDROID AND MINING APPLICATIONS

[1]Mettu Jhansi Rani, [2]Venkata Kondaiah Bedadam , 3 Ms. Nandagiri Kiranmai
[1,3] Department of Computer Science & Engineering, Bhoj Reddy Engineering College for Women, Hyderabad.
[2]Senior Software Engineer, Tech Mahindra, Hyderabad

Abstract— The rapid proliferation and ubiquity of mobile, smart devices in the consumer market has forced the software engineering community to quickly adapt development approaches conscious of the novel capabilities of mobile applications. The combination of computing power, access to novel on board sensors and ease of application transfer to market has made mobile devices the new computing platform for businesses and independent developers The development of millions of software applications for these mobile devices often called as 'apps'. Current estimates indicate that there are hundreds of thousands of mobile app developers. As a result, in recent years, there has been an increasing amount of software engineering research conducted on mobile apps to help such mobile app developers. In this paper, we discuss current and future research trends within the framework of the various stages in the software development life-cycle: requirements (including non-functional), design and development, testing, and maintenance. While there are several non-functional requirements, we focus on the topics of energy and security in our paper, since mobile apps are not necessarily built by large companies that can afford to get experts for solving these two topics. The recent advances done in these stages and then the challenges present in current work, followed by the future opportunities and the risks present in pursuing such research.

Keywords—Mobile apps, Mining app markets.

I Introduction

In the context of this paper, a mobile app is defined as the application developed for the current generation of mobile devices popularly known as smart phones. These apps are often distributed through a platform specific, and centralized app market. In this paper, we sometimes refer to mobile apps simply as apps. In the past few years we are observing an explosion in the popularity of mobile devices and mobile apps [17]. In fact, recent market studies show that the centralized app market for Apple's platform (iOS) and Google's platform (Android), each have more than 1.5 million apps [8]. These mobile app markets are extremely popular among developers due to the flexibility and revenue potential. At the same time, mobile apps bring a whole slew of new challenges to software practitionerssuch as challenges due to the highly-connected nature of these devices, the unique distribution channels available for mobile apps (i.e., app markets like Apple's App Store and Google's Google Play), and novel revenue models (e.g., freemium and subscription apps).

To date the majority of the software engineering research has focused on traditional "shrink wrapped" software, such as Mozilla Firefox, Eclipse or Apache HTTP [79]. However, researchers have begun to focus on software engineering issues for mobile apps. For example, the 2011 Mining Software Repositories Challenge focused on studying the Android mobile platform [90]. Other work focused on issues related to code reuse in mobile apps [84], on mining mobile app data from the app stores [34], testing mobile apps [70] and teaching programming on mobile devices [95]. Therefore, we feel it is a perfect time to reflect on the accomplishments in the area of Software Engineering research for mobile apps and to draw a vision for its future. Note that we restrict to just the software engineering topics for mobile apps in this paper, and even that not exhaustively due to space restrictions (we skip topics like usability or performance engineering since an entire paper can be written on each of these topics). We do not discuss the advancements in other areas of research for mobile apps such as cloud based solutions, or networking in mobile apps.

The purpose of this vision paper is to serve as a reference point for mobile app work. We start by providing some background information on mobile apps. Then, we discuss the current state-of-the-art in the field, relating it to each of the software development phases, i.e., requirements, development, testing, and maintenance as shown in Figure 1. We also talk about two non-functional requirements: energy use and security of mobile apps. Finally, even though it is not one of the software development phases, we talk about the software engineering challenges and recommendations for monetizing mobile apps. Along with a discussion of the state-of-the-art, we also present the challenges currently faced by the researchers/developers of mobile apps. Then we discuss our vision for the future of software engineering research for mobile apps and the risks involved, based on our experiences.

[1]Corresponding Author

Our hope is that our vision paper will help newcomers to quickly gain a background in the area of mobile apps. Moreover, we hope that our discussion of the vision for the area will inspire and guide future work and build a community of researchers with common goals regarding software engineering challenges for mobile apps. A word of caution though - the discussion of the current state-of-the-art is not meant to be a systematic literature survey (for a more comprehensive study please refer to Sarro et al. [88]), and the future directions of research are based on our opinions that have been influenced by our knowledge of the research in this community. researchers have begun to focus on software engineering issues for mobile apps. For example, the 2011 Mining Software Repositories Challenge focused on studying the Android mobile platform [90]. Other work focused on issues related to code reuse in mobile apps [84], on mining mobile app data from the app stores [34], testing mobile apps [70] and teaching programming on mobile devices [95]. Therefore, we feel it is a perfect time to reflect on the accomplishments in the area of Software Engineering research for mobile apps and to draw a vision for its future. Note that we restrict to just the software engineering topics for mobile apps in this paper, and even that not exhaustively due to space restrictions (we skip topics like usability or performance engineering since an entire paper can be written on each of these topics.

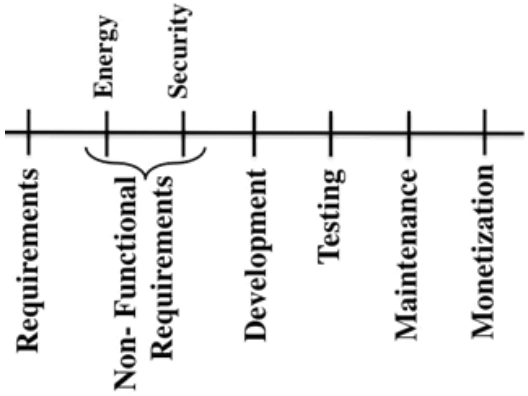

Fig. 1. A Framework For Presenting The State-Of-The-Art In Software Engineering Research For Mobile Apps.

The rest of the paper is organized as follows: Section II presents the necessary background information. Sections III-IX discusses the various software engineering research advance-ments made with respect to mobile apps. Section X concludes the paper.

II. Background

Mobile apps have been around for a long time now. Back in the 1990s they were usually created by device manufacturers like Nokia and Motorola. These apps performed certain basic tasks. Later on, wireless service providers started making apps to differentiate the devices sold on their network to others. At the same time, third party companies started making apps for the mobile platforms like the Windows mobile OS and the Symbian OS. These included games for the devices and other utility apps. However, there was no centralized place where end users could acquire these apps.

The most modern iteration of the mobile apps started in 2007, when Apple announced the first generation of the iPhones. At the same time Apple also announced the centralized market for mobile apps called the 'App Store', through which, the end users had to download all their apps. Soon after in 2008, Google deployed their own platform (Android) and their own app market the 'Android Market' (which was later renamed as 'Google Play'). Similar app markets were released for the mobile phone platforms developed by Microsoft, and BlackBerry as well. With these other app markets, now the mobile app developers have an even larger customer base to sell to. It is estimated that there are currently 2.6 Billion mobile phone users, who mostly own smart phones [59]. An overview of the various stakeholders in the world of mobile apps is shown in Figure 2.

With the introduction of app markets for each platform, now developers have the ability to manage the distribution of their software through one centralized market for each platform.

All developers big and small have the same app market, thus making it an even playing field for anyone to succeed. Also, the app markets made it easy for the developers to upload their apps, manage updates to them, and push the latest version seamlessly to the end users. Thus a combination of market potential, ease of use, and democratized platform, made it highly lucrative for developers to build mobile apps.

With the increased use of smartphones and mobile apps by end users, and development of these mobile apps by software developers, mobile apps became an obvious area for software engineering researchers to examine. One of the earliest software engineering papers on such mobile apps was the study of micro apps on the Android and BlackBerry platforms by Syer et al. [91], and one of the earliest studies on the app markets was by Harman et al. [34]. Since then, there have been plenty of studies on all sorts of data that can be mined from the app markets, with the app themselves being just one type of data.

We think the increase in such software engineering studies on mobile apps are because of two reasons - (1) since the app markets are publicly available, it is now possible to mine the data relatively easily (although later in this section we explore where researchers faced trouble in getting this public data), (2) a variety of new types of data that were previous not available are now available and reliably well linked together. Some of these new types of

data are discussed below (and a snapshot of the app store is in Figure 3).

The app markets are not just a venue for the developers to upload their app, and the user to download their app. App markets also have a rating and review system in place, where app users can describe their opinions on the app in free form text. The review data is rich in what users want from the app - both features and bug fixes, along with praise for the features that they love. Therefore, such review data has now become a treasure trove of data for requirements engineering researchers (more about this in Section III). Each of the reviews also have a numeric rating, which are then aggregated to determine the overall rating of the app, thus making it easy for users to know if the past users thought an app was good or not. Additionally, these numeric ratings also provided researchers with a clear way of knowing if an app is good or not, and if the review by an user is overall of a complimentary or derogatory nature. Therefore, researchers may only need natural language processing techniques like sentiment analysis to know which parts of a review was complimentary/derogatory of the app.

The app market also allows for the developer to post release notes on each of the app's versions. Researchers are able to mine this information to determine how the apps are evolving. Another piece of information available in the app store for each app is the contact information for the developer. Therefore, now researchers can contact app developers with anything interesting that they find about the app. We are also able to mine apps that are similar to the current app, and therefore examine how similar or different an app is from other apps.

Knowing the similarity between apps is further facilitated in the app markets by the category classification. Each app in the app store has to be classified in one of many predefined

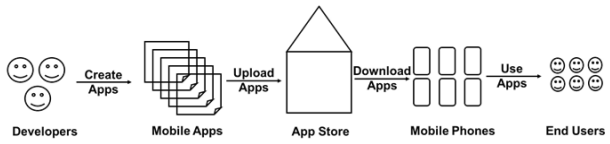

Fig. 2. Overview of the various stakeholders with respect to modern day mobile apps.

Fig. 3. Snapshot of an app in the app market

categories. Therefore, now as researchers we have access to apps that have been self reported to be in the same domain. This gives researchers tremendous potential to conduct research that can be controlled for the domain of the app. Often we see that a software engineering research study is done on an IDE, like Eclipse and another OSS project like the browser Firefox [20]. However, we do not know what domains of applications that these results transfer to. In the world of mobile apps, if we conduct our research on only game apps, then we can be more certain that our findings would apply to other game apps.

Additionally, all these various data points are available for hundreds of thousands of apps in a public facing website making it a rich dataset for researchers to crawl.

A. Common Challenges

In the next section, we discuss the accomplishments, challenges and risks for each of the development phases. However, one challenge seems to be a common challenge that impacts all of the development phases, public access to data. Such access challenges manifests in three ways.

Firstly, app stores restrict public access to their data and typically only allow for access to a subset of all the user reviews. For example, in the case of the Google Play store, one can only access 500 reviews for an app.

Secondly, app stores do not provide the source code of the apps, or any other associated artefacts like test code, or design and requirement documents. Only the app binary and release notes are made available.

Finally, with respect to the release notes, and the app binary, one can only get them for the latest release. There is no historical information that can be collected from the app store (except user reviews). The only way to gather

historical information on the various releases of the mobile apps is to continuously mine the app stores at regular intervals (like daily or weekly basis).

III. Requirements

A number of studies have focused on requirement extraction for mobile applications. Contrary to traditional work on software requirements, which mainly focused on analysis of the requirements and specifications document, the majority of mobile app-related studies leveraged app reviews posted by users to extract requirements. For example, Iacob et al. [40] used linguistic rules to detect feature requests from user reviews. Then, they summarize the feature requests to generate more abstract requirements. Galvis-Carreno and Winbladh [27] extract topics from user reviews in order to revise requirements. They show that their automatically extracted requirements match with manually extracted requirements. Guzman and Maalej [30], [60] use natural language processing (NLP) techniques to identify app features in the reviews and use sentiment analysis to determine how users feel about app features. They also compare their extracted features to manually extracted features and find that the extracted features are coherent and relevant to requirements evolution tasks.

Besides requirements extraction from the user reviews, there have been several studies on feature analysis. For example Rein and Munch [78] present a case study for feature prioritising. Finkelstein et al. [25] extract the set of features from the release notes available in the app store for a large collection of apps. They found a mild correlation between the number of features in an app and the cost of an app. Sarro et al. [89] examined feature migration lifecycles among apps.

Finally many previous studies have looked at the app reviews and tried to understand what complaints that users have about an app [26], [38], [41], [43], [67], [71]. In a previous study, we manually analysed and tagged reviews of iOS apps to identify the different issues that users of iOS apps complain about [46],

[49]. We hope to help developers prioritize the issues that they should be testing for.

A. Challenges and Future Directions

The fact that requirements are extracted from app reviews has its own challenges. In many cases and for many apps, there may not be enough user reviews or the quality of the reviews may be low. All of the aforementioned studies need a high quantity and quality of user reviews. Chen et al. have done some initial work in automatically identifying reviews that are informative [19]. However, there is still more work left to be done in this area. For example even if there are high quality reviews available, we do not know if we actually did get all the reviews from the app store [64]. Typically app stores restrict the public to be able to see only a subset of all the reviews. In the case of Google Play it is 500 reviews. In the case of the Windows Marketplace, they allow you to see as many as can be loaded in the page before the browser crashes. Therefore, we have a sampling issue, which has been illustrated by Martin et al. [64]. One interesting problem that has already been addressed by app markets like Google Play is the ability for the developer to reply to user reviews when they have addressed a requirement.

Another challenge is the applicability of the NLP techniques used to extract requirements from app reviews. However, off-the-shelf NLP tools are 1) not designed to extract software requirements and 2) not designed to analyze text from user reviews (which can be very brief, tend to be highly unstructured, and have typos).

Therefore the natural directions of research in the area of requirements engineering are as follows: building NLP techniques that are not subject to the limitations in the user reviews (and exploiting the newly available knowledge bases), come up with sampling techniques that takes the sampling bias into account, and building robust data collection tools that are able to collect a more complete set of reviews. All these research opportunities will allow us to mine requirements from the user reviews in a more efficient manner (as Maalej et al. state in their recent publication [97], the future of requirements engineering is data driven).

Some more recent research directions in requirements engineering are (a) prioritizing features that have been suggested by users. AR-Miner [19] has already scratched the surface of this

problem, by proposing a novel ranking algorithm to prioritize the groups of reviews identified. The authors also found that their prioritization was comparable to actual developers, and identifying traceability links between user reviews and app features, such as the tool CRISTAL [72]. However, it is important to know if all users are equal or are some users more influential and therefore, reviews by them might be more impactful to implement. Another complementary research problem is in determining which features should be dropped?

B. Risks

One of the risks involved in pursuing the above lines of research is that we may have reached the limits of NLP when analysing poorly written user reviews. Another risk is that maybe users prefer the features that they are provided before they ask for it, and when the user complains about the features, then it is already too late. The only solution might be to build an updated review system for the app stores that allows a better mechanism for feature requests from the users.

IV. Energy

Due to the fact that energy (or battery) is a scarce resource for mobile devices, a plethora of studies have proposed ways to measure and save energy of mobile apps. One of the first works related to the measurement of energy of mobile applications is GreenMiner by Hindle et al. [36], [37], which is a dedicated hardware platform that enables measurement of energy consumption of mobile devices. In other work, Halo et al. [33] propose a technique that leverages program analysis to provide per-instruction energy modelling. They show that their approach can estimate energy consumption to within 10% of the ground truth for Android apps. Liu et al. [57], present their tool Green Droid that will automatically identify the energy inefficiency bugs in Android apps. Similarly Banerjee et al. [15] detect energy bugs in mobile apps.

Other studies performed empirical research on energy consumption in order to provide developers with ways of minimizing it. For example, Pathak et al. [74] proposed a taxonomy of energy bugs based on more than 39,000 posts. They also propose a framework for the debugging of energy bugs on smartphones. Li et al. [52] perform an empirical study on 405 apps to better understand energy consumption. They make several interesting findings such as: 1) the majority of a mobile app's energy is spent in the idle state and 2) networking is the component that is more resource heavy. Linares-Vasquez et al. [54] present an empirical study into the categories of API calls and usage patterns that consume high energy. The findings of the empirical study can help developers reduce the energy consumption when using certain categories of Android APIs. Wan et al. [98] propose a technique that detects UI hotspots to help developers identify energy problems and reduce energy consumption. Linares-Vasquez et al. [55] propose a multi-objective approach that generates colour themes that optimize energy usage of mobile apps.

A. Challenges and Future Directions

The two main challenges in energy related research for mobile apps is not knowing what to measure for accurately identifying energy issues, and then trying to fix the issues for the developers. This is because the current state-of-the-art tools are not easily accessible to developers. Therefore, we need good estimates of energy use. In fact, there has been very little work on even understanding how much developers know about energy bugs [75], [73], and which of their actions actually cause them [87], [86]. Knowing more about developer coding habits and which ones cause more energy bugs could be impactful research.

Future directions in energy research could be in the area of identifying practical ways in which energy usage can be improved in apps. Another potentially impactful area of energy research is trying to understand how and when our findings translate to other platforms. Currently most of the energy research is happening on the Android platform. For example will the same third party libraries have a similar impact on the Windows or BlackBerry platform? If not, then can we build tools that can make recommendations to developers who are building cross-platform apps?

B. Risks

One of the more practical risks for researchers who want to pursue this line of research is: access to the hardware that can measure power or settle for software models that can be inaccu-rate [53]. There are some initial solutions, like the GreenMiner framework, that are available for researchers to remotely access the hardware resources for energy measurements [37]. Even when researchers have access, there exists the issue of sampling frequency. If the sampling frequency for energy measurement is longer than the time interval in which energy bugs occur, then there is a strong chance that the results are not consistent.

V. Security

A number of recent studies focused on the security of Android apps. A tangential line of work to this is the examination of permissions in mobile apps to prevent security vulnerabilities [23], [13]. However in security, there are two lines of research in the intersection of software engineering, mobile apps and security. The first line of work is in identifying vulnerabilities in apps. For example, Chin et al. [21] propose a tool called ComDroid, which detects communication vulnerabilities. Other work by Sadeghi et al. [85] proposed COVERT, a technique that detects inter-app vulnerabilities. Potharaju et al. look at various attack strategies and defence techniques from plagiarized mobile apps [76]. Quirolgico et al. present their work on how to vet mobile apps [77]. Jha in their PhD thesis catalogued a set of risks for mobile applications [42].

The second line of research is in finding malicious apps. For example Goral et al. [28] proposed the CHABADA tool, which detects unexpected behaviour of Android apps. CHABADA generates topics from app descriptions and compares the behaviour of the app against its description. The authors showed that CHABADA is effective in flagging 56% of malware

without any knowledge of malware patterns. In other work Avdiienko et al. [14] propose the Mudflow tool, which aims at detecting malicious apps. Mudflow examines the sources and sinks of data flows and examines if such data flows use sensitive data such as device ID and phone number. Then, Mudflow flags apps as being malicious if their data flows deviate from the data flows in benign apps. Arzt et al. [12] proposed the Flow Droid tool, which performs static taint-analysis of Android apps. Appscopy is a similar tool that detects Android malware through static analysis [24].

A. Challenges and Future Directions

Some of the well-known challenges that face static analysis of software, apply to the security research mentioned here as well. For example, it is well known that most static analysis approaches suffer from a high rate of false positives. That issue however, may be less critical for mobile apps since they tend to be smaller in size. Other work depends on data provided by the app developers, such as the app's description. Such approaches cannot guarantee to perform well for applications that do not have well documented descriptions.

Another challenge is the availability of data. Malicious code and vulnerabilities in code are ever changing (and at a great pace). In order to build techniques that can identify secure code from non-secure code statically, we need examples of both. However, there is a serious lack of malicious/vulnerable apps. Arzt et al. [12] built a publicly available benchmark suite of malicious apps called DroidBench. However, even this benchmark just contains 120 apps currently. Thus there is a real need to bolster this benchmark with more data.

There are many directions of future research that is possible in this area. The most obvious of this is to advance the state-of-the-art in static analysis research. When it comes to malware research the ultimate goal is to build a lightweight enough static analysis tool that can be deployed at the app store and prevent malicious apps from being uploaded to the store to begin with. Another outcome of such research is to provide the end user with an easy to use approach to understand what the app is doing and if its behaviour is abnormal.

Another more difficult research problem is to understand why developers are writing vulnerable code in the apps? How can we help them prevent unintentionally created security risks for end users? This line of research requires us to understand how to write secure software first. Then we need to be able to educate the developers. Meanwhile, can we build indicators to determine if an app will likely have vulnerable code in it or not?

B. Risks

Most of the work has been done for Android apps. This is mainly due to the fact that the Android platform is more open then other platforms, e.g., iOS or BlackBerry. Also the apps are written in Java for which there exists many decompiles and static analysis tools. Performing our studies on Android causes a risk in terms of how applicable the proposed approaches would work for mobile apps from other platforms.

Another risk is in just focusing on reactive approaches to security in order to solve the current security issues and not focusing on preventive solutions. Focusing on reactive approaches is not just an issue with mobile apps but with all software. However, with mobile apps due to the speed at which they are evolving, this issue could be even more potent - as we may never catch up.

VI. Development

While there has been quite a bit of past work in the areas of requirements, energy, security, testing and maintenance for mobile apps, there has been very little work that has been done on actually developing the apps. Most of the work has been from the platform developers like Google and Apple in providing the development tools required for building the mobile apps.

One of the earlier research papers in software engineering was by Syer et al. [91] who compared the source code of Android and BlackBerry applications along three dimensions, source code, code dependencies and code churn. They find that BlackBerry apps are larger and rely more on third party libraries, whereas, Android apps have fewer files and rely heavily on the Android platform. Hecht et al. [35] proposed a tool called Paprika to study antipatterns in mobile apps using their byte code. Khalid et al. [47] examined the relationship between warning from FindBugs and app ratings. They find that certain warnings correlate with app ratings. Cugola et al. [22] developed a declarative language for a specific type of mobile app. Around the same time, Tillman et al. developed Touch Develop, a platform to build mobile apps for the Windows Phone [96]. This platform was built to help novice developers with little to no experience in either software engineering or software development to build apps. Additionally, Acerbis et al.built the Web Ratio Mobile Platform for model-driven mobile app development [11], [4].

A. Challenges and Future Directions

With the popularity of all platforms increasing in the past few years, developers are tempted to develop the same app for multiple platforms (cross-platform development). In order to enable this, there are several frameworks that are available

Sencha, PhoneGap, and Appcelerator Titanium to name a few (some of the cross-platform development frameworks like Cocos2d, Unity 3D, and Corona are specifically for games). The developer has to build the app by only calling the APIs present in these frameworks, and at build time, an app for each platform is generated by the framework. However, all these frameworks, due to their design have an adverse affect on both the performance of the app and its user interface. Very little research has been conducted to help developers understand the costs and benefits of the various approaches of developing cross-platform apps [99].

This issue therefore, provides researchers with a tremendous opportunity to positively affect the developers.

Coming up with the next best cross-platform app development approach would be of very high impact.

B. Risks

With the platforms evolving as fast as they are to keep up with the competition, it may be very difficult to build a static solution for cross-platform development - the solutions must evolve just as fast. Additionally, there are hardware and app market policy mismatches that have to be taken care of. Even the study of the issues in cross-platform development, may be difficult because it may be difficult to link the apps across the app markets. Lastly, in some cases, mobile app developers may obfuscate their apps, making the study of their development a challenge since one would need to deal with the obfuscation of the code before being able to study the app.

VII. Testing

A wide range of studies have developed techniques to help mobile app developers improve the testing of mobile applications, in particular by trying to improve UI and system testing coverage. Hu et al. propose the Monkey tool, which automates the GUI testing of Android apps [39]. Monkey generates random events, instruments the apps and analyses traces that are produced from the apps to detect errors. Another tool proposed by Machiry et al. [61] is Dynordoid, which is a tool that dynamically generates inputs to test Android apps. Contrary to Monkey, Dynordoid enables the testing of UI and system events. Due to this difference, the authors showed that Dynordoid can achieve 55% higher test coverage compared to Monkey. Mahmoud et al. [62] presented the EvoDroid tool, which combines program analysis and evolutionary algorithms to test Android apps. The authors show that EvoDroid can outperform Monkey and Dynordoid, achieving coverage values in the range of 70-80%. Linares-Vasquez et al. [56] propose MonkeyLab, which mines recorded executions to guide the testing of Android mobile apps. While all these approaches are general purpose test generation approaches, Kim et al. [51] look at performance testing of mobile apps at the unit test level.

Different from the aforementioned work, another line of work aims to help developers deal with the Android fragmentation problem (i.e., the fact that Android has many devices). For example, Ham came up with their own compatibility test to prevent Android fragmentation problems [31]. Khalid et al. [48] proposed an approach that leverages user reviews to prioritize which Android devices an app should be tested on. Han et al. [32] examine device level fragmentation for the Android platform to understand vendor-specific bugs.

A. Challenges and Future Directions

One of the biggest challenges that researchers face in their current line of research on automated tests for mobile apps is that they are not able to achieve high code coverage [45], [80]. This is partially because of the inability to produce a wide range and variety of inputs and partially because of apps that are designed for user input (like game apps or apps that require a login), which cannot be automatically generated. Often the automated testing tools are unable to proceed down a certain execution path due to the inability to generate inputs, and therefore cannot test anything further along that execution path. Therefore research in generating a wider range of input that can mimic a human could have a great impact on automated mobile app testing tools.

Another challenge is that often researchers build tools that will work on the app binary since that is the only thing to which they have access. The availability of more OSS apps could yield in more robust tools. One repository of OSS apps is the F-Droid repository [2]. However, from past research we know that only a very small percentage of these apps are actually successful apps in the app market [93]. A repository of OSS apps with the corresponding app binaries made available as a benchmark suite could greatly help researchers in advancing the state-of-the-art in app testing. We would also like to point out that availability of successful OSS apps would advance the state-of-the-art in all areas of software engineering for mobile apps.

Currently most of the work as described above, focuses on automated testing of mobile apps. Even with these tools, the tests are often run on a single device and/or a simulator. However, with the increased success of multiple platforms there is now a large amount of cross-platform apps. Additionally, in all the platforms the apps need to run on different hardware with different versions of the OS (either due to different versions of the device, as in the case of the iPhone or on different devices, as in the case of the Android/Windows Phone platforms). Thus even if the app is tested on one device, there is no guarantee that it may work on another device. However, these problems are not entirely new. In the past software developers have had to develop for the PC/Mac/Linux platforms with varying hardware. While it is not a new problem, it still remains a challenge to test these apps across varying hardware and platforms. Thus one area of research with the potential for high impact is that of cross-platform testing. A recent study by Joorabchi et al. [44] describes a tool, CHECKCAMP that tests for inconsistencies between iOS and Android versions of mobile apps using extracted abstract models. Such a study is a step in the right direction, but a better understanding of cross-platform apps is still needed.

B. Risks

One of the big risks in pursuing the above line of research is that researchers may not have access to all the various devices and/or platforms. Additionally, there is no easy way to identify cross platform apps from the app stores. So

far, there has been no effort to build such a database of cross platform apps that researchers can analyse.

VIII. Maintenance

The area of software maintenance is one of the most researched areas in Software Engineering. However, due to the fact that mobile apps is a young subarea within SE, the maintenance of mobile applications remains to be largely undiscovered. Moreover, since mobile apps are different, the studies related to the maintenance of mobile apps tend to focus on issues that have not been traditionally studied in past software maintenance studies. For example, most mobile apps display advertisements, and as has been shown in prior studies, these advertisements require a significant amount of maintenance [82]. That said, a number of prior studies investigated the maintenance of mobile apps from different perspectives, e.g., code ruse and ad-related maintenance.

Mojica-Ruiz et al. [84], [81] compared the extent of code reuse in the different categories of Android applications. They find that approximately 23% of the classes inherit from a base class in the Android API and 27% of the classes inherit from a domain specific base class. Furthermore, they find that 217 mobile apps are completely reused by another mobile app. Syer et al. [93] compares mobile apps to larger "traditional" software systems in terms of size and time to fix defects. They find that mobile apps resemble Unix utilities, i.e., they tend to be small and developed by small groups. They also find that mobile apps tend to respond to reported defects quickly. Minelli and Lanza [68], [69] proposed SAMOA, a software analytics platform that was used to analyze 20 Android apps. Similar to Syer et al., one of their main findings is that mobile and tradition software are different since mobile apps tend to be very small in size, rely heavily on third-party libraries and essentially do not use inheritance. Bavota et al. [16], show that the quality (in terms of change and fault-proneness) of the APIs used by Android apps negatively impacts their success, in terms of user ratings. Similarly, McDonnell et al. [65], study the stability and adoption rates for the APIs in the Android ecosystem.

Another line of work examined Android-related bug reports. Bhattacharya et al. [18] study 24 mobile Android apps in order to understand the bug-fixing process. They find that mobile bug reports are of high quality, especially for security related bugs. Martine et al. [63] analysed topics in the Android platform bugs in order to uncover the most debated topics over time. Similarly, Liu et al. [58] detected and characterized performance bugs among Android apps.

A. Challenges and Future Directions

Some of the challenges in maintenance research for mobile apps is that often there is a lack of historical data. The software maintenance research community has greatly benefited from openly available artefacts like source control and bug repositories of OSS projects. They now have a large trove of data to evaluate their hypotheses on. Such a support has spurred an increased level of research in software maintenance as evidenced by the number of research publications on it. However, for the most part there are not many OSS mobile apps as discussed in the previous section. Most of the current research is based on the data available in the app markets. Therefore, with limited fine grained commit level information it is difficult to conduct maintenance research.

One interesting line of future research is in estimating the maintenance cost for a mobile app. Currently there are just anecdotal estimates [3]. Careful studies and insight into this can greatly help small time mobile app developers to plan ahead. While traditional maintenance issues like bug localization may not be an issue due to the small size of the apps, mobile app developers would like to be able to triage features and bugs from the user reviews (as seen in Section III).

Finally as mentioned in Section IX, there are several companies that collect operational data from mobile apps that have been installed on millions of devices. Most of these companies provide the app developers with the data and some rudimentary analysis on them. There is a wide variety of reliability and performance problems that can be solved by building tools and approaches that mine such operational data (past work has barely scratched the surface of such a problem by looking more at the server side of mobile applications than the client side [92]).

B. Risks

From past research we have seen that mobile apps are small [93], and have very quick release cycles [66]. With such rapid release, it may be the case that the maintenance effort might overlap a lot with the evolution effort. Hence, it may not be easy to identify costs pertaining to maintenance. Additionally, the variety of apps are far more than the variety of successful desktop applications. For example, a small recipe app like AllTheCooks [5], and a large application like Microsoft Excel [7] are equally popular, but they may have completely different maintenance efforts. Thus the issue of placing the results in the right context becomes paramount. Therefore, it is highly recommended to keep track of the app domain when conducting maintenance case studies.

IX. Monetization

Some of the successful gaming apps (like Angry Birds and Candy Crush) and productivity apps (like Microsoft Excel) are produced in established software development companies with large teams. However, from past work, we know that successful apps can be developed by one or two core developers too [93]. In such apps where the

development organization is small, often the developers will also have to make several engineering decisions that could affect their bottom line. Therefore software engineering researchers have examined how we can provide data to mobile app developers so that they can make these decisions in a more careful fashion. Some of these research studies are presented below.

Past research has found that ratings and downloads are often very highly correlated [34]. Additionally, Kim et al., also found ratings to be one of the key determinants in a user's purchase decision of an app [50]. However, Ruiz et al. examined the rating system in Google Play and found that the current rating system of cumulative averages across all versions of an app makes the ratings sticky and thus does not encourage developers to improve their app [83].

While ratings may not be a sufficient condition for more downloads, it may be a necessary condition. With more downloads, the developers stand to increase their revenues as well. This is because, often mobile apps are just monetized through in app advertising. The app itself is given at no cost. If more users use an app, more ads are shown to users, and more revenues are generated for the app. A more detailed overview of the various stakeholders with respect to mobile ads can be found in the work by Ruiz et al. [82]. In the past, we have found the number of advertising networks that a developer connects to does not impact the user perception of an app (ie the rating) [82]. There were apps that used as many as 28 different ad libraries, and still had a good user rating. This was probably because, even though the developers connect to many different networks through many different libraries, they still were displaying just one ad at a time. Thus we recommended, that a mobile app developer could add as many ad libraries as they wanted (as long as they did not disrupt the user experience) without impacting the rating. However, we found that including particular ad libraries could affect the rating. This was because, the ad libraries were being intrusive, and the user perceived the app to be intrusive as well. Hence, we recommended that the developer be careful about what ad networks they were connecting too. This study [82], is a good example of how software engineering researchers could make software related recommendations that could improve the monetization strategies of an app.

Additionally we looked at the cost incurred by the users when using a free app with advertisements in them [29]. We found that there are considerable energy, network and performance related costs associated with ads. Thus we recommend that users be careful when using an app with ads. If users do realize this, then developers should be ready with an alternative that has no ads (which could be paid).

Currently there are also several analytics companies (AppAn-nie [1], Quetta [9], Criticism [6] etc.) that provide valuable usage data to developers for improving their monetization strategies. They track the downloads of apps, and how the apps are being used, when users purchase things from the app etc. These companies are able to track such user data, by incorporating tracking libraries in the mobile devices. Using this information developers are able to make smarter data driven decisions with respect to making their app more successful. However, most of these recommendations are more from a marketing perspective than software engineering perspective.

A. Challenges and Future Directions

Even though, it finally comes down to the amount of money made through an app in most cases, we as software engineering researchers care more about what makes an app successful. Success can mean different things to different people. It could mean more downloads, it could mean driving more users to a business that is outside the mobile space, and it could mean just recognition by means of having a high rating. Thus success is not just one fixed measure, but one from a set of possible measures depending on the context.

Depending on the choice of success measure, researchers can then come up with various hypothesis for what factors could affect this success measure. By gathering a set of possible factors (independent variables), and the success measure (dependent variable) for a large collection of apps from the app store, we can then model the data to see when an app can be successful. We can also see what factors are most related to the success measure, and then carry out controlled experiments to see how far the correlations translate to causation. There has been some recent initial work in this direction where Bavota et al. [16] looked at the impact of using certain APIs on the ratings and Tian et al. [94] model a set of factors (like size of app, complexity of app and its UI, quality of the library code etc.) against the ratings. They were able to find that there is initial evidence that high rated apps have a certain DNA (certain value for various factors). In the future, we need to come up with more such factors to be evaluated, and strengthen our current findings with user studies. These factors can be derived through mobile app user and developer surveys for example.

B. Risks

While there is tremendous potential in determining under what circumstances an app will be successful, there are certain risks too. It will be easy to misinterpret the correlation in the data that we gather as causation. We need to be rigorous in controlling for other factors like category of apps, date apps were released, platform they run on etc. We also need to identify these factors based on common sense intuition and motivation based on previous

work. It would be easy to correlate the name of an app with the success of the app and conclude that app names starting with a particular letter are more successful. However, we should avoid such pitfalls and only model factors that an app developer/past research would actually consider as a factor that could affect the app success.

We also need to be careful about placing our results in the correct context. Results that are obtained from free apps may not translate to freemium apps (apps with in app purchases) or paid apps. Freemium apps are those apps where the app developer gives away the app for free and then charges for additional features or content inside the app. The paid monetization model is the traditional model, where the app developer sells the app directly to the user for a monetary price. There are has not been much software engineering research on the freemium/paid apps since they are more difficult for researchers to get access to. Given this limitation we simply do not know how results would generalize. Another challenge caused by the lack of access to historical data, is the fact that success of an app can change overtime, e.g., initially one might want to just have a popular app, but later will look for revenues, however it is difficult to capture such changes.

X. Conclusion

In conclusion, we believe that due to popularity of mobile apps, and the impact that research can have on developers from both small and large organizations, combined with the abundance of publicly available data, interesting research opportunities still left to be explored, and a vibrant community being built around it, software engineering research for mobile apps is a great place for young researchers to start.

References

[1] App Annie, howpublished = Online: https://www.appannie.com/, year = Last accessed Oct 2015, source = https://www.appannie.com/,.

[2] F-Droid, howpublished = Online: https://f-droid.org/, year = Last accessed Oct 2015, source = https://f-droid.org/,.

[3] How much would it cost to make and to maintain (operating costs) an instant messaging app like whatsapp?, October 2015.

[4] Model-driven development based on omgs ifml with webratio web and mobile platform. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, Engineering the Web in the Big Data Era, volume 9114 of Lecture Notes in Computer Science. 2015.

[5] Allthecooks recipies. Online: https://play.google.com/store/apps/details?id=com.mufumbo.android.recipe.search&hl=en, October Last accessed Oct 2015.

[6] Crittercism. Online: http://www.crittercism.com/, Last accessed Oct 2015.

[7] Microsoft excel. Online: https://play.google.com/store/apps/details?id=com.microsoft.office.excel&hl=en, October Last accessed Oct 2015.

[8] Number of apps available in leading app stores as of july 2015. On-line: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, Last accessed Oct 2015.

[9] Quettra. Online: https://www.quettra.com/, Last accessed Oct 2015.

[10] Shonan meeting on mobile app store analytics. http://shonan.nii.ac.jp/seminar/070/, Last accessed Oct 2015.

[11] Roberto Acerbis, Aldo Bongio, Stefano Butti, and Marco Brambilla. Model-driven development of cross-platform mobile applications with webratio and ifml. In Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems.

[12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.

[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In Proceedings of the 2012 ACM Conference on Computer and Communications Security.

[14] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In 37th IEEE/ACM International Conference on Software Engineering, Florence, Italy, May 16-24, 2015, Volume 1, pages 426–436, 2015.

[15] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

[16] Gabriele Bavota, Mario Linares Vasquez,´ Carlos Eduardo Bernal-Cardenas,´ Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of Android apps. IEEE Trans. Software Eng., 41(4):384–407, 2015.

[17] Berg Insight. The mobile application market. Online: http://www.berginsight.com/ReportPDF/ProductSheet/bi-app1-ps.pdf, Last accessed Oct 2013.

[18] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering.

[19] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In Proceedings of the 36th International Conference on Software Engineering.

[20] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. Explaining software defects using topic models. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories.

[21] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys ’11, pages 239–252, 2011.

[22] Gianpaolo Cugola, Carlo Ghezzi, Leandro Sales Pinto, and Giordano Tamburrelli. Selfmotion: a declarative language for adaptive service-oriented mobile apps. In Proceedings of the ACM SIGSOFT 20th