Analysis of Feature-Completeness in Android Cross-Platform Frameworks

Andreas Biørn-Hansen[1], Tor-Morten Grønli[1], and Siri Fagernes[1]

[1]Mobile Technology Lab, Department of Technology, Westerdals Oslo School of Arts, Communication and Technology, Norway

Abstract

In cross-platform mobile development research, we frequently encounter mentions of limitations and constraints potentially imposed by technical tools and development frameworks. This is especially prominent in the context of programmatic device- and platform feature access, including features such as GPS, Internet and device camera access. Although the majority of the literature does not empirically validate these claims, they have reached acceptance in both practitioners' communities and academic research. By downloading a sample of 300,000 Android applications available on the Google Play Store and analysing them, we set forth to find which platform- and device features are the most commonly included in deployed apps. Based on the results, we map the features to their availability in five major cross-platform development frameworks, thus provide an overview of feature completeness and potential shortcomings in these popular frameworks. Our findings indicate that the scrutinised frameworks range from 86.37% to 95.46% feature-completeness and can thus facilitate the development of mobile apps relying on features that are commonly found in our assessed sample of Android apps.

1 Introduction

With projected revenue streams approaching 200 billion USD in year 2020 [1], software applications – or *apps* – targeting smartphone devices is a tremendous source of potential income for mobile developers and businesses. However, taking part of the app economy is a complicated endeavour, involving a myriad of decisions, both technical and business in nature. An initial decision might be: which mobile operating system(s), henceforth referred to as *platform(s)*, must be targeted to reach the majority of the app's target group? According to a study by Francese *et al.* [2], developers find that developing for multiple platforms is challenging for reasons involving inherent differences in platform features and application programming interfaces (APIs), code maintenance, and testing, among a variety of reasons. This may not be surprising, as following traditional app development methods, developing an idea into an app that is executable on both Android and iOS will require two completely separate codebases. Thus, development efforts tend to double compared to developing for only one platform [3, 4]. On a similar note, does the development team's competency include mobile development-specific knowledge? This typically includes proficiency in the programming languages Java or Kotlin for

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

Android, Swift or Objective-C for iOS, and C# for the Windows platform, in addition to understanding of platform-specific guidelines on user experience and visual design [5, 2]? In fact, for each platform added to the product specification, an additional set of programming language(s), design guidelines, third-party libraries, architectural patterns and more adds to the development complexity. Thirdly, should the app make use of platform- and device functionality, including such as the GPS module, Internet access for HTTP calls, file system access for storing and retrieval of files, and have the ability to intercept incoming SMS? Existing literature frequently refer to this type programmatic access as *access to device and platform features* [6], and is a commonplace requirement in decision trees- and frameworks (e.g. [7, 8, 9]).

Indeed, developing platform-specific apps – terminologically referred to as *Native* apps – for smartphones may be severely complex, and reportedly so, both costly and time consuming [3, 4]. However, practitioners and researchers have since the advent of the smartphone tried to minimise the required effort of developing apps that can run on multiple platforms, i.e. have the ability to share large portions of code between otherwise heterogeneous platforms. This is typically achieved through the use of abstractions layers, whose purpose is to abstract platform-specific functionality into models and more generic APIs, that will – depending on the platform onto which an app is deployed – execute the respective functionality in a fashion that is understood by the underlying platform [4]. Both in research and practice, these abstraction layers are commonly referred to as *cross-platform frameworks* [10], and are seen as replacements or supplements of the traditional Native app development approach. The *cross-platform* umbrella term embodies a wide array of technical solutions and overarching development approaches and has in recent years seen backing from major companies including Facebook, Google and Telerik advocating the possibilities of cross-platform development through their own portfolio of solutions including React Native, Flutter and NativeScript respectively. The purpose of these frameworks is to deliver a mostly homogenous and platform-unified development experience. By doing so, these frameworks help developers to create mobile apps without having to write platform-specific code for business logic and user interface from scratch twice if an app is to be available on both Android and iOS. To exemplify, a cross-platform framework can enable developers to write business logic in JavaScript, which acts as the abstraction layer, then the framework can interpret the JavaScript upon runtime and route API calls to their respective functionality in the platform's native programming language.

From reviewing and assessing existing literature on cross-platform development, we frequently encounter criticism (e.g. [11, 9, 7, 12]) towards the frameworks for potentially imposing restrictions and constraints on app developers, especially so when cross-platform frameworks are compared to the traditional platform-specific Native development approach. A frequently criticised part of cross-platform development frameworks is their ability to take advantage of platform- and device features (e.g. [9]), i.e. that they do not sufficiently deliver programmatic access to such as the device's contact list, camera, GPS and so on, features that are vital for the creation of complex apps. This often unbacked criticism of technical frameworks is our motivation for this current research. We are interested in whether or not such frameworks can in fact make use of platform- and devices features typically included in already-deployed apps. Thus, for our experiment we downloaded 300,000 Android apps which have all been deployed to the Google Play app store. Downloading Android apps for analysis was made possible by the AndroZoo repository, a massive database containing more than 5,8 million Android installation files (APKs) [13]. We then extracted the list of permissions from each app,

a permission typically representing a platform- or device feature the user must grant an app access to use, e.g. reading of incoming SMS or access to modifying device settings. The extracted dataset was then processed, leaving us with a list of commonly requested permissions, or *features* as they become in our context. These features were then mapped against their availability in five popular cross-platform development frameworks, namely Ionic Framework, React Native, NativeScript, Xamarin.Forms, and Titanium [14]. Our findings indicate that all five frameworks are in fact capable of accessing and programmatically exposing the absolute majority of the most common features extracted from the 300,000 apps, thus seemingly contradicting frequently encountered claims in previous research.

The rest of this paper is organised as follows: The upcoming section is devoted to related work, in which we present related research and studies we build on and discuss in relation to. In section 3, our research question is presented along with an overview of the research design. Our findings are then presented in section 4, and further discussed in section 5. Lastly, section 6 concludes the paper and provides directions for further work.

2 Related Work

Cross-platform mobile development has been subject to scholarly research since the popularization of the smartphones and app stores, with early work including studies by Heitkötter *et al.* [6], Kramer *et al.* [15], and Charland and LeRoux [16]. The nature of these articles range from descriptive to technical, some aiming to communicate the technical possibilities and shortcomings of cross-platform frameworks (e.g. [6, 15]), while others (e.g. [16]) discuss more overarching topics including performance and user experience of cross-platform apps. In common, they all discuss the importance of device- and platform features, and to a varying degree criticise the facilitation of access to such features in cross-platform development frameworks. In more recent years, mobile app stores have enjoyed the attention of numerous research projects [17], and mining and analyses of their content has previously led to interesting findings also in the context of cross-platform app development. There has been a slight shift in methodology, as where previous work has often been descriptive and holistic, research on cross-platform development now increasingly incorporate analyses of ultra-large software repositories such as the Google Play Store [18].

To help better understand the value and presence of cross-platform built apps in the Google Play store, a total of 11,917 Android apps were analysed by Malavolta *et al.* [19]. They found that (i), based on app store reviews, cross-platform apps were valued similarly to traditionally developed Native apps, and (ii) that cross-platform apps tend to request access to the same Android permissions as Native apps. They also state that development frameworks need more work to better support platform- and device features, which is the investigative goal for our current study. Thus, we build on the work presented in [19], and further investigate where they left off, specifically targeting Android permissions.

A sample-wise much larger study is presented by Viennot *et al.*, communicating results from the analyses of 1,1 million Android apps [20]. The focus of the study is split between an architecture proposal for the crawling and analyses of apps, and (i) the rating of app-pairs, i.e. apps that exist in both Google Play Store and Apple's App Store, (ii) the use of advertising platforms, cross-platform development frameworks and similar third-party libraries, and (iii) the presence of secret service tokens in decompiled source code. While they do not focus on the analysis of permission usage, their study is, to the best of our knowledge, the largest of its kind in terms of sample size. Thus, we can draw

from their work in terms of architecture and tools, e.g. their proposed used of certain software for the decompilation and processing of Android apps.

We also find relevant studies of non-technical nature, including previous work carried out by Biørn-Hansen, Grønli and Ghinea. Their study assessed 14 implementation-oriented academic studies as an effort to identify which platform- and device features are commonly included in applied research on cross-platform mobile apps [8]. With the data presented in our current study, we can discuss the appropriateness of the findings and recommendations presented in [8], and whether or not the studies traversed in fact do experiment with features commonly included in real-world deployed apps. Their findings indicate that a variety of device sensors including proximity, accelerometer and GPS are typically included in research on cross-platform apps, along with Internet/network access, camera access, and device file system access. We further discuss these findings in the context of our own, as presented in section 4.

3 Research Design

Throughout this section, we elaborate on the steps and design making up the research method. This starts with AndroZoo URL transformation through APK downloading, extraction and parsing of the AndroidManifest permission file, to finally the algorithm used for counting thus measuring frequency of permission usage. The technical part of the study involves a series of scripts and tools to gather and analyse the Android APK files. The flowchart depicted in Figure 1 illustrate the process further elaborated upon in the upcoming subsections.

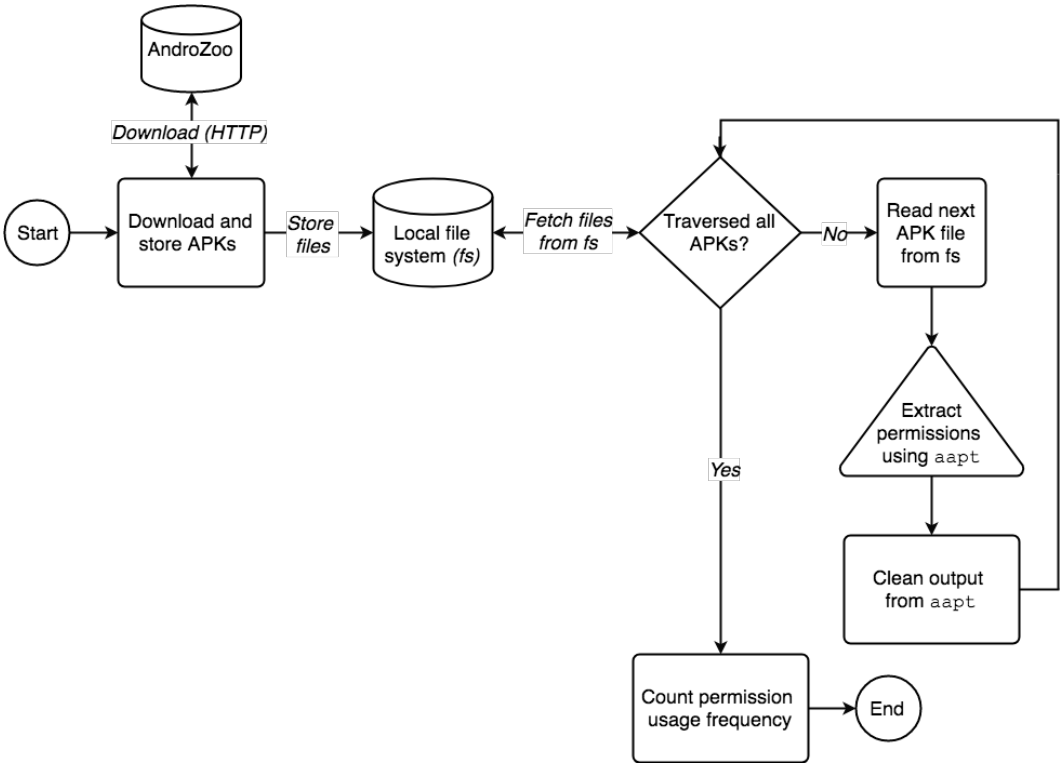

Figure 1: Overview of application extraction and analyses process

From our case and the assessment of existing literature as laid out in the introduction and related work, we often-so find mentions of how cross-platform development frameworks are unable to compete with the Native development

approach in the context of programmatic device- and platform feature access. With this as our core motivation and with a mapping of features implemented in apps deployed to Google Play Store, we pose the following research question:

> **RQ:** How feature-complete are the top five cross-platform mobile development frameworks?

Downloading Application Package Files

An alternative to AndroZoo would be to scrape the Google Play Store directly, however, as described by Li *et al.* [13], this process involves a series of steps to circumvent restrictions imposed by the app store to hinder such scraping. Thus, to avoid unnecessarily dealing with such restrictions, we opted to leverage the AndroZoo service, which goal is to cater to researchers studying deployed Android apps. AndroZoo provided a comma-separated values (CSV) file containing metadata for more than a million APK files[1]. Structure-wise, the CSV contained the following column headings: `sha256, sha1, md5, dex_date, apk_size, pkg_name, vercode, vt_detection ,vt_scan_date, dex_size, markets` [13].

The `sha256` value was the unique primary key used to identify a particular APK file and was further used in the download process. We extracted all the `sha256` values from the CSV and built a URL string for each app listed in the spreadsheet. The in total 300,000 line-separated URLs were saved to a text file. The APKs were all downloaded from the AndroZoo [13] service using the command-line interface (`aria2` [21]), allowing for configuration of parameters including maximum concurrent downloads, and the input file containing the line-separated download URLs. The `aria2` tool provided us an out-of-the-box solution for concurrency management, resuming of downloads, and more. This was particularly important, as AndroZoo limits concurrent downloads to 30.

Extraction and Parsing of Application Permissions

An APK's list of permissions reside in its AndroidManifest.xml, a metadata configuration file which is bundled together with all of the assets, resources and source code making up the app. It was crucial that the pace of the permission list extraction was rapid, as even an extraction time of one second per APK would result in more than 83 clock hours, given that the method was indeed sound and did not end in failure at any point during the task.

Two tools for extracting permissions were evaluated, including Apktool and Apkanalyzer. While Apktool provided the most comprehensive suite of features, allowing for the extraction and decompilation of source code and manifest file, it did not match with our requirement of sub-second extraction time. We found that the official Android Apkanalyzer tool, part of the Android Asset Packaging Tool suite, or `aapt` for short, provided a command line interface tool for easy extraction of app metadata, including APK package name and a list of permissions, both of which were saved for analyses.

During the permission extraction stage, `aapt` was unable to parse the AndroidManifest of 61 APKs, resulting in a final dataset of 299,939 analysable apps. This dataset was then cleaned, involving the removal of permissions that were included more than once per manifest file.

[1]CSV file from AndroZoo: `https://androzoo.uni.lu/lists`

Counting Permission Usage

The final step of the analysis process was the counting of permission usage, i.e. finding how frequent a permission is encountered in the AndroidManifest.xml files traversed. The permission counting was implemented using an algorithm based on the MapReduce model. Thus, we build a data structure of the encountered permissions where each `permission` is accompanied by a `counter` property which is incremented at every encounter of the permission upon traversal of the dataset.

4 Results

To avoid reporting on less frequently used permissions, we set the minimum limit of occurrences in the dataset to 16,000, or 5.33% of the initial dataset (300,000). We removed six permissions from the dataset, four due to API deprecation and two more were removed due to API changes / deprecation.

Permissions

The encountered permissions are listed in Table 1 accompanied by their number of occurrences in the traversed dataset.

Table 1: # occurrences of permissions in the analysed apps' manifest files

Permission	**Occurrences**
*android.permission.*INTERNET	283 763
*android.permission.*ACCESS_NETWORK_STATE	262 937
*android.permission.*WRITE_EXTERNAL_STORAGE	203 987
*android.permission.*READ_PHONE_STATE	144 840
*android.permission.*ACCESS_WIFI_STATE	132 221
*android.permission.*WAKE_LOCK	129 690
*android.permission.*VIBRATE	101 920
*android.permission.*ACCESS_COARSE_LOCATION	98 687
*android.permission.*ACCESS_FINE_LOCATION	93 659
*com.google.android.c2dm.permission.*RECEIVE	67 451
*android.permission.*GET_ACCOUNTS	66 753
*android.permission.*RECEIVE_BOOT_COMPLETED	60 276
*android.permission.*READ_EXTERNAL_STORAGE	54 915
*android.permission.*CAMERA	49 694
*android.permission.*SYSTEM_ALERT_WINDOW	43 038
*android.permission.*CHANGE_WIFI_STATE	33 097
*android.permission.*RECORD_AUDIO	31 406
*android.permission.*CALL_PHONE	29 921
*com.android.vending.*BILLING	29 684
*android.permission.*WRITE_SETTINGS	26 208
*android.permission.*READ_CONTACTS	24 747
*android.permission.*MODIFY_AUDIO_SETTINGS	16 209
*android.permission.*SEND_SMS	16 195

Feature-mapping

We employed the following structure when querying for feature availability in a given framework: `"{PERMISSION_TAG or FEATURE_NAME} + {FRAMEWORK_NAME}"`. Upon encountering potentially relevant search results, we systematically ensured that the feature(s) made available through the inclusion of the permission was in fact programmatically available from within the cross-platform development environment. This was indeed necessary, as permissions can be included in the AndroidManifest.xml file for use in the Native part of the cross-platform app, whilst the features provided by the permissions may not be exposed through the cross-platform framework itself (i.e. not accessible for the cross-platform app developer to make use of). For our study, we are only interested in the results that could confirm or disconfirm a feature's programmatic availability in the set of technical frameworks, either implemented into the framework itself, or being available through third-party plugins. In terms of frameworks, those listed as part of Table 2 were chosen for scrutiny due to prevalence in industry outlets and in newer research on cross-platform development [8, 14].

The order of Table 2's content is identical to that of Table 1. Instead of listing permissions, as we do in Table 1, we have translated each permission into their respective device- and platform feature, e.g. `ACCESS_COARSE_LOCATION` from Table 1 is listed as "Access Geolocation" in Table 2.

Table 2: Mapping of features against their availability in cross-platform frameworks

Feature *Ordered as Table 1*	**Frameworks**				
	Ionic	React Native	Native-Script	Xamarin	Titanium
Internet Access	✓	✓	✓	✓	✓
Access Network State	✓	✓	✓	✓	✓
Write to File System	✓	✓	✓	✓	✓
Read Phone State	✓	✓	✓	✓	✓
Access WiFi State	✓	✓	✓	✓	✓
Wake Lock	✓	✓	✓	✓	✓
Vibrate	✓	✓	✓	✓	✓
Access Geolocation	✓	✓	✓	✓	✓
Receive Push Notifications	✓	✓	✓	✓	✓
Get Accounts	✓	✓	~	✓	✓
Receive Boot Completed	✓	✗	✓	✓	✓
Read from File System	✓	✓	✓	✓	✓
Camera	✓	✓	✓	✓	✓
System Alert Window	✗	~	✗	~	✗
Change WiFi State	✓	✓	✓	✓	✓
Record Audio	✓	✓	✓	✓	✓
Call Phone	✓	✓	✓	✓	✓
In-App Billing	✓	✓	✓	✓	✓
Write Settings	✓	✓	✓	✗	✓
Read Contacts	✓	✓	✓	✓	✓
Modify Audio Settings	✓	✓	✗	✗	✓
Send SMS	✓	✓	✓	✓	✓

5 Discussion

Investigating our dataset, we can look at the data from the two analysed perspectives. Firstly, in relation to permissions we see that the majority of apps, i.e. more than 50%, rely on Internet connectivity, access to listen on changes to a device's network state, and the ability to read and write to the device storage. These findings align with those reported by Biørn-Hansen, Grønli and Ghinea in their assessment of feature inclusion in academic research on cross-platform app development [8]. Thus, for future applied research on app development, findings both from this current study as well as those previously reported in [8] should provide a point of departure in terms of which features to include to achieve generalisability and validity in both academia and practice.

Secondly in terms of feature mapping, it interesting to see that the majority of the features included for assessment are in fact programmatically exposed to the cross-platform development environments through the respective development frameworks. As such, developers opting for a cross-platform development approach are likely to find that the device- and platform features they may be accustomed to from the Native development approach, are in fact very much available. Revisiting some of the related work assessed in section 2, our findings are a step towards better understanding of cross-platform frameworks' facilitation of access to platform- and device features, as discussed by Malavolta *et al.* [19]. However, our research does not take into account the availability of less-used features. Thus, for apps that rely on custom features, e.g. third-party SDKs lacking support for the frameworks we have assessed, our results might not be applicable.

Table 2 also report of certain features that we did not manage to find included in the frameworks scrutinised. We found that the System Alert Window permission was not directly accessible from within any of the cross-platform framework environments. This permission is what allows apps such as Facebook Messenger to draw on top of other apps and thus always be visible to the user, i.e. how they made the infamous overlaying chat heads user interface on Android [22]. While this arguably could be implemented in native code and exposed to the cross-platform code environments, we identified no such efforts. However, React Native's in-app debugger window relies on the permission, but we did not find any mentions of the debugger window being able to draw on top of *other* apps than itself. We also identified a Xamarin.Android project[2] with the ability to overlay other apps, alas – it was not implemented in the Xamarin.Forms cross-platform framework, as these are two different development environments. Thus, we marked both React Native and Xamarin with the symbol ~, representing implementation plausibility.

It is important to note that while all the frameworks could possibly be capable of supporting the features marked with ✗ or ~ in Table 2, we did not manage to identify any official or third-party efforts to achieve such implementations. However, NativeScript's documentation boasts the framework's extensiveness, as they state that all platform APIs can be accessed from a JavaScript environment [23]. Nevertheless, and based on our reported findings – in the event of a product specification requiring superimposed and "always visible" user interface elements, the Native development approach is to the best of our knowledge the only feasible approach.

[2] https://github.com/LifeCoder45/chatheads-xamarin

Finding: The feature-completeness of the frameworks assessed range from 95.46% to 86.37%. The two top-ranking frameworks are equally feature-complete, being Ionic and Titanium, both of which were identified to support all but the System Alert Window feature.

6 Conclusion and Further Work

Our motivation for conducting the research at hand was the frequent encounter of unbacked claims regarding constraints imposed by cross-platform mobile development frameworks. We set out to verify the actuality of these claims through the extraction and aggregation of permission schemes from 300,000 deployed Android apps. The permissions were translated to device features, e.g. programmatic access to device camera, then we mapped the results to the availability of the features in five cross-platform frameworks. Our findings indicate that the majority of the most-used features identified in the dataset are in fact exposed and made available through the assessed frameworks. These results contradict claims put forth by previously identified research on the subject and may be of assistance in decision-making and to further bring forth the possibilities of cross-platform app development as an alternative to the Native development approach.

As the technical part of the study involved numerous individual scripts and tools, we aim to work towards a complete toolchain for downloading, extraction, and analyses of primarily Android applications. Our hypothesis is that through the development and deployment of such a toolchain, large-scale Android app analyses and experiments can be conducted with less overhead in terms of code and third-party tools. With the potential of millions of files that must be decompiled, traversed, analysed and transformed, computational performance is of high importance. Thus, for further work we would also put emphasis on evaluation of execution environments and the performance of existing tools developed for tasks including extraction and decompilation of APKs into code that resemble source code.

Acknowledgement

We wish to acknowledge the important and exhaustive work of the AndroZoo research group at the University of Luxembourg, making it possible to mass download Android APK files without scraping the Google Play Store.

References

[1] App Annie. Mobile app revenues 2015-2020. `https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/`, November 2016. Accessed: 2018-4-19.

[2] Rita Francese, Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Mobile app development and management: Results from a qualitative investigation. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 133–143. IEEE, July 2017.

[3] Henning Heitkötter and Tim A Majchrzak. Cross-Platform development of business apps with MD2. In *Design Science at the Intersection of Physical and Virtual*

Design, Lecture Notes in Computer Science, pages 405–411. Springer, Berlin, Heidelberg, June 2013.

[4] Raj Rahul and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference*, pages 625–629. IEEE, December 2012.

[5] Tor-Morten Gronli, Jarle Hansen, Gheorghita Ghinea, and Muhammad Younas. Mobile application platform heterogeneity: Android vs windows phone vs iOS vs firefox OS. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 635–641. IEEE, May 2014.

[6] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Comparing cross-platform development approaches for mobile applications. In *Proceedings 8th WEBIST*, pages 299–311. SciTePress, April 2012.

[7] Mounaim Latif, Younes Lakhrissi, El Habib Nfaoui, and Najia Es-Sbai. Cross platform approach for mobile application development: A survey. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, pages 1–5. IEEE, March 2016.

[8] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Baseline requirements for comparative research on Cross-Platform mobile development: A literature survey. In *Proceedings of the 30th Norwegian Informatics Conference*. Bibsys, November 2017.

[9] Mohamed Lachgar and Abdelmounaïm Abdali. Decision framework for mobile development methods. *International Journal of Advanced Computer Science and Applications*, 8(2):110–118, 2017.

[10] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating Cross-Platform development approaches for mobile applications. In *Web Information Systems and Technologies*, Lecture Notes in Business Information Processing, pages 120–138. Springer Berlin Heidelberg, 18 April 2012.

[11] Luis Corral, Andrea Janes, and Tadas Remencius. Potential advantages and disadvantages of multiplatform development Frameworks–A vision on mobile environments. In *Procedia Computer Science*, volume 10, pages 1202–1207. SciVerse ScienceDirect, 9 August 2012.

[12] Mounaim Latif, Younes Lakhrissi, El Habib Nfaoui, and Najia Es-Sbai. Review of mobile cross platform and research orientations. In *2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS)*, pages 1–4. IEEE, April 2017.

[13] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. AndroZoo++: Collecting millions of android apps and their metadata for the research community. *arXiv [cs.SE]*, September 2017.

[14] Sacha Greif, Raphaël Benitte, and Michael Rambeau. Mobile & desktop frameworks. `https://stateofjs.com/2017/mobile/results`, December 2017. Accessed: 2018-3-1.

[15] Dean Kramer, Tony Clark, and Samia Oussena. MobDSL: A domain specific language for multiple mobile platform deployment. In *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, pages 1–7. IEEE, November 2010.

[16] Andre Charland and Brian LeRoux. Mobile application development: Web vs. native. *Queueing Syst.*, 9(4):20, April 2011.

[17] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847, 2017.

[18] University of Luxembourg. AndroZoo publications. `https://androzoo.uni.lu/publications`. Accessed: 2018-4-16.

[19] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '15, pages 56–59, Piscataway, NJ, USA, 2015. IEEE Press.

[20] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, volume 42, pages 221–233, New York, NY, USA, June 2014. ACM.

[21] Tatsuhiro Tsujikawa. aria2. `https://aria2.github.io/`, 2017. Accessed: 2018-4-12.

[22] Keval Patel. Create chat heads like facebook messenger. `https://medium.com/@kevalpatel2106/create-chat-heads-like-facebook-messenger-32f7f1a62064`, November 2016. Accessed: 2018-4-17.

[23] Deyan Ginev and Nikolay Tsonev. Accessing native APIs. `https://docs.nativescript.org/core-concepts/accessing-native-apis-with-javascript`, January 2018. Accessed: 2018-4-19.