Tampere University

Jarkko Saarinen

EVALUATING CROSS-PLATFORM MOBILE APP PERFORMANCE WITH VIDEO-BASED MEASUREMENTS

Faculty of Information Technology and Communication Sciences
M. Sc. thesis
Supervisor: Timo Nummenmaa
April 2019

ABSTRACT

Jarkko Saarinen: Evaluating cross-platform mobile app performance with video-based measurements
M. Sc. thesis
Tampere University
Master's Programme in Computer Science
April 2019

Smartphone sales are nowadays centered around two platforms: Apple's iOS and Google's Android. These two platforms are vastly different and generally a native app made for one platform can't be used on the other, which means that organizations have to develop two separate apps to reach customers on both platforms. Several cross-platform mobile app development tools have been created to address this issue by allowing developers to write the app once and have the tool work as an intermediary that makes the app run on both platforms. These tools generally work by using workarounds and creating abstractions on top of native SDKs, which can cause performance overhead. This study investigated the performance of apps created with these kinds of tools when compared to native apps.

To test the performance of apps created with these tools, a benchmarking app was implemented with five different cross-platform development tools and the native development tools of Android and iOS. The tests measured how fast apps could perform tasks like opening a new screen and reacting to a button press. Collecting measurements that are comparable between apps created with different tools was done by adopting a method previously used to test input lag in games. This method involves recording a video of the device running the test and then the video is analyzed frame by frame. The videos were captured using a high-speed video camera and screen recording software.

The results showed that the cross-platform apps often have some areas where they perform worse than their native counterparts, especially on Android. These problematic areas included app launch times, moving between screens and displaying a list of items. The performance disadvantages however weren't generally significant enough to make using cross-platform tools a bad choice for organizations looking to reduce their app development costs, but some attention needs to be paid when selecting which tool to use.

Keywords: Mobile apps, cross-platform, performance analysis

Contents

1. Introduction 1
2. Related work 4
3. Cross-platform mobile app development patterns 6
   - 3.1. Hybrid app pattern 6
   - 3.2. Interpreted app pattern 7
   - 3.3. Cross-compiled app pattern 8
4. Cross-platform app development tools 9
   - 4.1. Cordova 9
   - 4.2. React Native 10
   - 4.3. Titanium 11
   - 4.4. Xamarin 12
   - 4.5. Flutter 13
5. Comparing performance between tools 15
   - 5.1. Logging-based measurement 16
   - 5.2. Video-based measurement 17
   - 5.3. Testing setup 20
6. Benchmarking app 22
   - 6.1. General implementation guidelines 22
   - 6.2. Landing screen 24
   - 6.3. Button reaction delay screen 27
   - 6.4. Large list of items screens 28
   - 6.5. Heavy computation screen 29
   - 6.6. Vibration screen 30
7. Results 31
   - 7.1. Comparing high-speed camera and screen recording software results 31
   - 7.2. App launch times 34
   - 7.3. Moving between screens 35
   - 7.4. Reacting to a button press 38
   - 7.5. Heavy computation 38
   - 7.6. Triggering vibration 40
   - 7.7. Frozen frames during scroll 40
8. Discussion 42
   - 8.1. General results 42
   - 8.2. Evaluating the tools 44
   - 8.3. Video-based performance measuring 45
9. Conclusion 49

References 51

1. Introduction

Developing mobile apps is hard and time consuming. To make matters worse, an app created for iPhones can't run on Android phones and vice versa. To bring an app to both major smartphone platforms, app developers need to create it two times basically from scratch. That means that there's twice the code to write and maintain and twice the resources needed to create the app.

There are several ways to address this issue. First is to only create the app on one platform and ignore the other. Depending on the audience the app is targeting this might be feasible. For example, Android's market share in Asia was 84,12% in September 2018 [MOSMSA, 2018], so for apps targeting Asia it can make sense to focus only on Android. On the other hand, iPhone users are more likely to spend money on apps and in-app purchases [Nelson, 2018], so it's understandable that paid apps often focus on iPhones. The second approach is to skip creating an app completely and focus on the web. Modern mobile browsers let web apps use lot of phone's features such as location and gyroscope and users can "install" the web app by creating a shortcut to it on their phone's home screen, making this an appealing approach to organizations that have already created a solid web version of their app and are looking into the mobile market.

The third approach is to use cross-platform mobile app development tools. These tools let app developers to write their app code once and then run it on multiple platforms, in this case iOS and Android. Some notable tools include Apache's Cordova, Microsoft's Xamarin and Facebook's React Native. The main appeal over the web app approach for most of these tools is that they give developers access to even more of the phone's features, such as flash storage and running code native to the platform. The way these tools work on the surface level is that they create an API that abstracts the features of native platforms, and then behind the scenes do platform specific actions. The developer then writes the app with the abstracted API and the heavy lifting of all the platform specific code is done by the cross-platform tool.

The history of cross-platform smartphone app development tools is as old as iPhone app development, with QuickConnect for iPhone starting development in spring 2008 soon after the iPhone SDK was first released to initial testers [Barney, 2008]. QuickConnect and most early cross-platform tools use webviews, the part of native mobile SDKs that embeds a web browser in the app, to run web apps on the phone. These webview-based apps can also be called hybrid apps as they leverage both web and native technologies [Raj and Tolety, 2012]. PhoneGap, a webview hybrid tool currently known as Apache Cordova, started development in fall 2008 and is still widely used. Solutions that weren't based on webviews started appearing next year with Appcelerator Titanium adding Android and iOS support in summer 2009 [Krill, 2009]. Since then many tools have come and gone, such as Adobe Air, Sencha Touch, jQuery Mobile and Intel XDK.

Some game engines, like Unity, have also added support for mobile platforms but are out of scope of this thesis.

The app developer community has taken great interest in cross-platform tools. In 2013 Business Insider estimated that 10% of smartphones had Appcelerator Titanium based apps installed on them [Bort, 2013]. In July 2018, 0,95% of all questions posted on Stack Overflow had “react-native” tag on them (As a comparison, “android” tag was present in 6,2% and “ios” in 2,3% of all questions during that same month) [SE_API, 2019]. Some large companies using cross-platform tools at least in parts of their apps include Alibaba, Google [FShowcase, 2019], Facebook, Microsoft (Skype), Pinterest and Uber [WURN, 2018]. Facebook, Google and Microsoft are also behind some the currently more popular cross-platform tools, React Native, Flutter and Xamarin respectively. Several scientific papers have also been published about these tools, which will be discussed more in Chapter 2 about related works.

Even though cross-platform tools have lots to offer, they are a controversial topic in the app developer community. Developers in native tools focused Internet communities, such as Android developer subreddit, can even act a bit hostile against them [IXSTB, 2018; WDTSHF, 2018] or upvote harsh jokes about them [Illbuyajuicer, 2018]. Some of them see cross-platform tools as inferior and are annoyed that their popularity might eventually affect their jobs [Empiricalis, 2018]. Especially hybrid apps seem to be hard to optimize for performance as searches like “Cordova performance” yield lots of questions on Stack Overflow, with answers varying from modifying app metadata files [Gregavola, 2012] to using third party libraries or non-default embedded browsers [Kevin, 2016]. Other common complaints are bad tooling [FJRRN, 2018], larger app sizes due to packaged runtimes [Mudiyala, 2017] and not supporting all platform specific features [LouisCAD, 2016]. Most cross-platform tools have a mechanism to add platform specific code to help access platform specific features, but that somewhat defeats the purpose of using a tool whose main selling point is to have single codebase run on all platforms.

The main topic of this thesis is to see what the state of cross-platform development tools was in 2018 and if there still were any basis on the complaints on their bad performance. The research included designing a benchmarking app and implementing it with native Android SDK, native iOS SDK and 5 different cross-platform tools. The performance testing involved trying out measuring time from videos recorded from the device, something not previously done in this context. These videos were recorded in two sets, with the first one being recorded with a high-speed video camera and second one with screen recording software. The thesis includes discussion on the previously used timing based on logging [Corral *et al.*, 2012; Willocx *et al.*, 2015] and why it might not fit this use case, and how well the video-based timing worked out.

The next three chapters will give brief introductions to previous studies on cross-platform tools, the principles that these tools are often based on and the tools used in this

study. Chapters 5 and 6 will go through how the comparison study was conducted by going through how performance of apps was measured and how the benchmarking app behaved. The results are given in Chapter 7 and Chapter 8 has more insightful discussion and thoughts about the results and the study. Chapter 9 wraps up with some final conclusions.

2. Related work

As smartphones and apps have taken a major part in people's lives in the last ten years, researchers have taken interest in the field as well. Papers about mobile cross-platform tools can be roughly divided to four categories: Ones that find and define different patterns on how to create cross-platform apps, ones that compare different tools and their performance, ones that try to understand the implications of their usage for developers, and ones that focus on the end user experience of cross-platform apps. Many papers have sections from more than one category, with most of them having at least a small comparison between different tools.

Raj and Tolety [2012] defined four different types of cross-platform apps: web apps, hybrid apps, interpreted apps and cross-compiled apps. Web apps use web technologies and are used through a web browser. Hybrid apps are also created with web technologies, but use a browser embedded in to a native app by the developer or a framework like PhoneGap. Interpreted apps use a runtime interpreter which lets the developer access platforms native APIs, including UI, through abstraction layers. Cross-compiled apps have parts of them written in languages that can be compiled to native binaries of each platform. The UI and platform specific functionality of the app is then written with the platforms native SDKs that can access these binaries. Xanthopoulos and Xinogalos [2013] defined the generated app pattern, in which apps are written in a domain specific language (DSL) of an app generator tool. The generator uses the DSL source code to generate native source code for each platform the generator supports. App generators at the time mostly supported data-driven apps with CRUD operations, but there weren't any widespread tools [Xanthopoulos and Xinogalos, 2013]. Taneja *et al.* [2016] defined virtual machine, distributed computation and hardware-based approaches with the virtual machine approach being an umbrella term for both hybrid and interpreted apps defined by Raj and Tolety [2012]. In distributed computation the app on user's device is just a dumb UI that delegates all or most computations to a remote server. When discussing the hardware-based approach, Taneja *et al.* [2016] mostly listed ways how mobile processors can theoretically be made more powerful and energy-efficient but didn't really give any examples what this approach means from an app developer's perspective.

The year 2012 saw several publications that compared different tools. Ohrt and Turau [2012] compared 9 different tools and checked which platforms they supported, what features they had and how well they performed. Their findings were that most of the tools had issues with developer support and resource usage (e.g. CPU, RAM) that needed to be addressed before they could be considered a serious alternative to native tools. Corral *et al.* [2012] created an app with both native Android SDK and PhoneGap hybrid app tool and compared executions times of several tasks between them. PhoneGap lost in all tasks but retrieving network information (and won that by only 0.1 milliseconds), struggling especially in file reading and accessing contact list. More recent studies, such as ones from Willocx *et al.* [2015] and Ferreira *et al.* [2018] have confirmed that cross-

platform apps can still use more resources but are catching up with native tools. Ciman and Gaggi [2017] took a unique approach to the subject and measured apps' energy consumption and found out that cross-platforms apps consume more energy than native apps. In some cases, cross-platform apps can perform better than native ones in some tasks like navigating between screens [Willocx *et al.*, 2015]. Ahti *et al.* [2016] even had their hybrid app performing better than a native Android app in app startup time and RAM usage due to native app requiring libraries to perform its tasks and be compatible with older versions of the Android platform.

Some of the papers comparing tool performance also have brief mentions about the developer tooling and experience, such as Ohrt and Turau [2012] listing IDE features, debugging and emulator support in their table of features supported by the tools. Heitkötter *et al.* [2013] focused their comparison on seven infrastructure oriented (such as licenses & cost, distribution methods) and seven developer oriented (such as maintainability, scalability) criteria. Their conclusion was that if native app like UI isn't necessary, using PhoneGap to create a hybrid app is the recommended approach as using web technologies brings a lot of benefits.

Studies that focused on user experience and impressions of cross-platform apps often had a test user group try out multiple versions of an app created with different tools. Studies that had users try the different app versions in quick succession had users mostly prefer the native versions [Humayoun *et al.*, 2013; Angulo and Ferre, 2014]. Andrade and Albuquerque [2015] had users first use either native or hybrid version of the app for two weeks and give feedback on it. After the two-week period there was another two-week period, but half of the users had their app version switched. Only 8 of 60 testers noticed performance differences between the two app versions, suggesting that performance differences between hybrid and native apps aren't too noticeable in everyday usage. One major exception in research method is a study by Mercado *et al.* [2016] which studied user feedback from apps published to Google Play and Apple App Store. On both platforms cross-platform apps were more likely to get complaints about metrics like performance, reliability and usability than native apps. They even saw that when Facebook app was changed from hybrid app to native the amount of complaints it got reduced.

This thesis mostly focuses on learning how different kinds of cross-platform tools work and the performance of apps created with them. The developer point of view won't be focused on as many points are either hard to analyze (e.g. how does one compare completeness of documentation between tools) or depend on personal preferences (e.g. does one like JavaScript or C# more). As it's very time consuming to have enough people test out the same app implemented with 6 tools on two different smartphone platforms to do any meaningful analysis, no end user feedback analysis was conducted.

3. Cross-platform mobile app development patterns

Previous studies have defined several patterns on how cross-platform smartphone apps are created. This chapter will take a deeper look on how these patterns are used to achieve cross-platform compatibility. The next chapter will introduce some tools that utilize these patterns.

One common aspect for all cross-platform patterns is that while they run some parts of the app in a cross-platform environment, they still need a native portion of the app to be able to run on a device. Depending on the pattern the native portion of the app can be just a container that makes the app installable and runnable on a device or an integral part of how the apps UI and features are implemented. In most cases, some mechanism is needed for the native and cross-platforms parts of the app to communicate with each other. In this thesis the terms "bridge" and "bridging" are used to generally address the communication mechanisms used in different patterns and tools.

As web apps and hybrid apps work fundamentally the same through a browser [Raj and Tolety, 2012], only hybrid apps will be focused on. The biggest differences between the two are the distribution method and the fact that web apps can't access native APIs [Raj and Tolety, 2012], both of which could cause problems in the comparison study later as the tests assume the app to be installed locally on the device and to have access to native APIs. There aren't any widespread tools that use generated [Xanthopoulos and Xinogalos, 2013], distributed computation or hardware-based cross-platform app patterns [Taneja *et al.*, 2016] so they won't be covered in this thesis. However, it doesn't mean that aren't any implementations of these patterns. For example, server driven UI approach used by Airbnb [Peal, 2018a] can be considered as an implementation or evolution of the partially distributed computation pattern described by Taneja *et al*. [2016].

3.1. Hybrid app pattern

Hybrid apps are created with web technologies and are run on a browser embedded in a native app [Raj and Tolety, 2012]. The Android SDK offers the WebView class to do this and the iOS SDK offers WKWebView (the older UIWebView is now deprecated). Normally these classes are intended to show content from the Internet but also support loading content packaged in the app, which is what enables hybrid apps to work. The embedded browser offers everything you'd expect from a normal browser: rendering HTML which can be styled with CSS and possibility to run JavaScript code to drive the application logic.

As Android's WebView and iOS's WKWebView are based on different browsers (Chromium and WebKit), minor differences in HTML rendering and supported JavaScript features are possible and even expected. Some of the differences in JavaScript features can be normalized by packaging polyfills (implementations of JavaScript APIs)

as part of the application or by using a transpiler that transforms source code that uses newer JavaScript features to more backwards compatible source code.

What makes these embedded browser apps “hybrid” is that they can access the platforms native APIs through the embedded browser [Raj and Tolety, 2012]. For example, the Android WebView class has methods “evaluateJavascript” and “addJavascriptInterface” which lets the application’s native Java portion call JavaScript functions and JavaScript to call Java methods respectively. This is essentially hybrid app version of bridging native and cross-platform portions of the app.

Tools for creating hybrid apps have two main benefits for developers over setting up an app with embedded browser by hand: they have tools to package the web app into a native app without writing any native code and provide premade bridges with the native code for commonly used native platform APIs. Often these tools also let developers add their own bridges if they want to or need some specific functionality from the native side of the app. More advanced tools can also support features like fetching updated source code and assets for the packaged web app without updating the app through App Store or Google Play (for an example of this, see https://github.com/Microsoft/cordova-plugin-code-push).

3.2. Interpreted app pattern

Interpreted apps work similarly to hybrid apps in that they use a separate runtime to run the app logic. The main difference between them is that interpreted apps use the native SDK tools for rendering [Raj and Tolety, 2012] instead of a browser’s HTML rendering engine. The interpreter runtime can be anything that can run code and communicate with the native SDK, like a JavaScript or Python interpreter.

Because rendering is not handled inside the runtime where developer code runs, developers can’t directly manipulate what is shown on the screen like in hybrid apps where they can update the HTML and CSS. This means that the cross-platform tool needs to provide developers some other means to describe the screen contents and then map that to platform’s native UI widgets. The mapping from the tool’s abstracted UI description to native UI widgets lets the cross-platform tool to change the look and feel of the app depending on the platform, making it closer to a native app experience [Raj and Tolety, 2012]. For example, let’s say that a developer tells the tool that they want to show a button with text “Hello world”. The tool can map that on Android to a Material Design themed rectangle with a drop shadow and all-capitalized text and on iOS to a clickable text with first letters of words capitalized.

Interpreted apps communicate with native APIs with bridges similar to how hybrid apps do. With the UI being rendered in the native portion of the app but defining what should be rendered and handling input events happening in the interpreted portion of the app, the bridges are a central part of interpreted app tools instead of just being an optional

way to access more device features like in hybrid apps. The bridging mechanism needs to be fast enough to not be a bottleneck in the apps' execution, as that can result in UI updates or reacting to user input being delayed, making the user experience worse. The implementation of bridges can be dramatically different between tools, even if they use a similar cross-platform runtime. For example, React Native allows developers to register native packages from which the JavaScript portion can call methods from [NM, 2018], while Axway's Hyperloop module allows Titanium apps to use any native code with no setup or glue code required [Hyperloop, 2019].

3.3. Cross-compiled app pattern

Cross-compiled apps use native binaries to deliver the cross-platform portions of the app [Raj and Tolety, 2012]. App developers create their applications with a compiled language and the cross-platform tool then compiles the code for each platform and packages it in an app. Using a compiled language and binaries can give these apps performance benefits over apps relying on interpreting their source code [Raj and Tolety, 2012].

Need for bridging between portions of the app depends on the target platform and the cross-platform implementation. Android apps run in Google's implementation of Java Virtual Machine (JVM), meaning that bridging through Java Native Interface (JNI) is needed to access APIs that aren't part of the Android Native Development Kit (NDK), like the UI. On iOS native apps are already compiled into machine code, so if the cross-platform tool supports communicating with Objective-C directly then no extra bridging or setup is needed. In practice this means that the tool would need to be either C or Objective-C based, and tools created with other types of technologies would need to use some bridging to enable interoperability between native iOS code and cross-platform code.

Raj and Tolety's [2012] original description of cross-compiled apps listed the need to recreate the UI on every platform as their main disadvantage. This means that at the time cross-compilation was more of way to share business logic code than to write complete cross-platform apps. Nowadays modern cross-compilation tools, like Xamarin with its Xamarin.Forms toolkit, can include UI tools similar to what interpreted apps traditionally have with abstracted UI definitions being mapped to native UI widgets by the tool [UTXMP, 2017]. Also at least one cross-compilation tool, Flutter, creates its UI with its own rendering engine backed by OpenGL or Vulkan [FSA, 2017], which makes it work with the same principle as cross-platform 3D game engines do. This means that UI events don't need to be sent from native code to cross-platform code but making the app look and feel like a native app on each platform means that the native UI widgets need to be recreated with Flutter's rendering engine.

4. Cross-platform app development tools

Most of the patterns from Chapter 3 require both native and cross-platform portions of the app to be aware of each other and to co-operate. At minimum the native portion needs to set up the cross-platform portion, and for the simplest hybrid app with no need to access native functionality that's all that's really needed. For any more complex apps though, bridging events like app lifecycle changes or requesting a link to be opened in a browser are needed. Creating the native setup code and cross-platform enabled abstractions for bridging the events can be cumbersome, so most of the time creating a cross-platform app means using a tool that does all the heavy lifting and allows developers to focus on the application specific code. Maybe one exception to this is the Raj and Tolety's [2012] original type of cross-compilation when not used to create most of the app in cross-platform code.

In this chapter we'll look at five selected tools that abstract most of the native SDKs away and allow developers to write basic apps with no need to write any native code. These five tools were used in the comparison study, which will be covered in following chapters. The two main criteria for being selected were that the selected tools should all represent different cross-platform patterns and that they should be at least somewhat relevant in the developer community in 2018. The "relevancy" of a tool can be quite vague, but here it was measured by looking at how many questions related to it were posted to Stack Overflow during summer 2018 (this was done by using the site http://sotagtrends.com/). With these criteria, Cordova, Flutter, React Native and Xamarin were selected. The fifth tool, Titanium, was added as it was featured in many previous studies on cross-platform app performance and thus could make comparing results with those studies easier.

4.1. Cordova

Apache Cordova, formerly known as PhoneGap, started development in 2008 and is one of oldest smartphone cross-platform tools [Johnson, 2008]. It allows bundling HTML5 apps as native apps, and thus is a hybrid app development tool. PhoneGap was originally developed by Nitobi Software, but when Adobe acquired Nitobi in 2011, the PhoneGap source code was given to Apache Software Foundation to create open source Cordova project [AAATAN, 2011]. Adobe still maintains PhoneGap as their own distribution of Cordova and offers extra tooling through their cloud services. Newest versions of Cordova support creating apps for Android, iOS, Windows and MacOS [PS, 2018].

Creating apps with Cordova doesn't require developers to write any native code or use native SDK tools. Instead the Cordova CLI tools are used to create and manage projects, and it handles packaging the app in a native SDK project and generating the glue code for it to show the packaged web app in a webview.

Bridging the native SDK code and web app code is done with Cordova's custom API, which wraps the webview's own bridging mechanism. Native methods that can be called are added to the project with plugins, which can be installed with the Cordova CLI tools. Plugins include all needed native SDK code and XML configuration files that Cordova uses to create objects needed to use the API. To call plugin functions, Cordova's bridging API takes the name of the plugin, the name of an action to run in the plugin and an array of parameters needed to run the action. Calls to plugins are asynchronous, and any results from the plugin action are delivered with callbacks.

As Cordova just wraps web apps into a native app, the UI in Cordova apps is written in HTML and CSS. Developers are free to use any coding styles or techniques that generally work on mobile browsers. Cordova documentation though suggests that developers should adopt the Single Page Application (SPA) pattern so that the Cordova JavaScript framework and other assets need to be loaded only once [NS, 2018]. Cordova doesn't include any built-in tools to make the app look like a native app on each platform, though their documentation includes a short list of external tools for styling web apps and a note that considering each platforms UI guidelines is important.

Cordova is a mature tool with documentation covering its APIs, how-to instructions and best practices. Debugging code is supported with Safari and Chrome debugging tools and other tools like Ripple. While not a common use case, Cordova documentation also includes instructions on how to integrate Cordova hybrid app portions into otherwise native SDK created apps on Android and iOS.

4.2. React Native

React Native is a JavaScript based interpreted cross-platform app development tool created by Facebook. It shares its main principles and some source code with ReactJS, which is a JavaScript library for creating web apps. It was originally created at a hackathon event at Facebook and its initial release was in January 2015 [Chedeau, 2015]. Facebook officially maintains support for Android and iOS platforms, but the community has created tools to run React Native apps on other platforms like Windows, macOS and browsers [OOTP 2018]. React Native is open sourced and currently uses the MIT license. Before February 2018 it was licensed with the BSD license with an additional clause that the license would terminate if the tools users were to sue Facebook for patent infringements [UTMITL, 2018].

React Native uses an asynchronous bridge to send messages between native and JavaScript portions of the app [Alpert, 2018]. The native portion of the app can communicate with the JavaScript portion by either sending events that JavaScript can register to listen to by name, or by sending properties (which basically are parameters used to describe the app's state) to the root node of the virtual DOM tree. JavaScript can send named events with optional callbacks to preregistered native modules like in

Cordova. Registering modules is done with macros on iOS and by passing them to ReactInstanceManager on Android. Facebook is planning to heavily rework the bridge, aiming to make it simpler and faster and to include at least some synchronous call support [Alpert, 2018].

Even though React Native is based on ReactJS, a web app framework, it doesn't use a webview to render the app. The way how ReactJS works is that it collects the developers' description of the app's UI into a virtual DOM (Document Object Model, a tree structure that most browsers use to organize HTML) and uses that to update the browser's actual DOM efficiently only in parts that change [RE, 2019]. React Native takes the same idea of a virtual DOM, but uses it to update a view hierarchy of a mobile platform's native SDK UI widgets instead of a browser DOM. Because of the mapping from a virtual DOM to native UI widgets, React Native apps can look like native apps on all platforms. Developers can create their own UI widgets by developing them in native code and then registering them as native modules. The UI is written in JSX, a HTML-like markup language which is converted into JavaScript when the app is bundled [JSXID, 2019]. JSX is written in JavaScript code files, so the UI and logic portions of the application aren't written separately like traditionally in the web where separate HTML, CSS and JavaScript files are often used.

React Native's tools and documentation are extensive, including support and tutorials for creating apps without using any native SDK tools, using some custom native code or integrating React Native into an app created mostly with the platform's native SDK. A lot of JavaScript and ReactJS guides and community resources also apply to React Native. While the ReactJS and React Native APIs have started to stabilize, there are still occasional breaking changes [RNChangelog, 2019].

4.3. Titanium

Titanium is another interpreted cross-platform tool that lets developers to create Android, iOS and Universal Windows Platform (UWP) apps with JavaScript. It's one of the older tools of its category and has supported Android and iOS since summer of 2009 [Krill, 2009]. It is free to use and open-sourced with Apache 2 license. It was originally developed by Appcelerator, but they were acquired by Axway in 2016 [Haynie, 2016]. Axway offers some additional premium features to Titanium developers, like geofencing and SQLite database encryption, in its paid premium licenses.

In Titanium bridging calls between native and JavaScript is done through modules and proxy objects. A module is a singleton proxy that defines an API, while normal proxies can be instantiated through the module. Proxy methods can take an array of parameters and return values synchronously, or they can asynchronously fire events to JavaScript code. Views are defined as special view proxies. In native code modules and proxies are defined and registered by extending Titanium SDKs Proxy classes and doing

some other platform specific setup (annotations, specific file naming schemes). Titanium also has a separate Hyperloop module that enables calling any native code without separately defining proxies. Hyperloop's documentation doesn't include information on how it achieves this. A tool with a similar feature, NativeScript, collects metadata of all available native APIs during build time using reflection which are then exposed to JavaScript code [WIARFNS, 2018]. A Stack Overflow answer from 2016 by an Appcelerator employee suggests that Hyperloop might work on a similar principle as he talks about a "hyperloop-metabase" that does reflection [Knöchel, 2016].

The currently recommended way to create app UIs in Titanium is using Alloy, an MVC framework that uses XML for defining layouts and a CSS-like language for styling. The XML can also bind functions as event listeners like in HTML. The "classic" method of creating UIs had developers create and manipulate the view proxies in JavaScript more directly. Axway is currently working on experimental Titanium ports of Vue and Angular, popular JavaScript web app frameworks.

Titanium and related Axway products (Alloy, cloud services) have good API documentations and most JavaScript guides and code that don't manipulate the DOM should also work with Titanium. The documentation doesn't mention any support to integrate Titanium to an existing app created with native SDKs. While Titanium isn't brought up in developer community discussions as often nowadays as other tools discussed in this thesis, it was mentioned in most related work papers.

4.4. Xamarin

Xamarin is a .NET based cross-compilation tool that lets developers use a common C# codebase to create Android, iOS and Windows apps. It was originally based on MonoTouch and Mono for Android technologies which Xamarin had licensed from Attachmate [SAXP, 2011] after Attachmate had laid off the teams behind them [Allen, 2011]. Microsoft bought the Xamarin company in 2016 [Guthrie, 2016]. Microsoft has open-sourced Xamarin with the MIT license.

The base Xamarin platform and apps created with it follow Raj and Tolety's [2012] original description of cross-compiled apps very closely: it lets developers to write the common parts like logic in a shared codebase that is compiled for each platform, and the UI is created separately for each platform with their native SDK tools (XML for Android, Storyboards for iOS). The separate Xamarin.Forms toolkit branches off from the original cross-compilation pattern by adding the option to create the app's UI with cross-platform tools too. The UI framework of Xamarin.Forms works similar to how interpreted apps' UIs work, as the layout and styles are defined in abstract XML which is then mapped to native UI widgets in runtime. In this thesis' comparison Xamarin is used with Xamarin.Forms as other tools are also used with their cross-platform UI creation tools.

Xamarin.Forms UIs are written in XAML (Extensible Application Markup Language), a superset of XML that is used to declare UIs in .NET applications. On top of declaring how the UI should be laid out, XAML can also be used to bind the UI to code (e.g. button click listeners, binding a value of a property to UI text).

A .NET embedding feature that lets an app built with native SDK tools call C# code was published in 2017. While Xamarin documentation doesn't seem to advertise it, using Xamarin.Android and Xamarin.iOS to write a bit of platform specific glue code, .NET embedding can be used to use Xamarin.Forms views in an otherwise native app [Montemagno, 2017]. Otherwise the documentation is exhaustive with API references, examples and even free courses. Being based on C# and .NET there's also a great supply of general resources and community made code modules that developers can use.

4.5. Flutter

Flutter is a unique take on the cross-compiled app pattern created by Google. It features its own OpenGL rendering engine that is part of the Flutter runtime, meaning that it doesn't need to use a bridge to native SDKs to create and interact with the app's UI like lot of cross-platform apps do. It supports Android and iOS and is also the UI framework for Googles Fuchsia operating system. It is open-sourced with the 3-clause BSD license. It was first demoed at Dart Developer Summit 2015 with the name "Sky" [Seidel, 2015]. Google has really started campaigning about Flutter in 2018 with 11 sessions at Google I/O [IOSchedule, 2018] and regular videos on Google Developers YouTube channel.

Flutters take on cross-compiled app pattern with its own UI framework and rendering engine comes with benefits and downsides. Some benefits are better performance due to not using bridging as much, the apps looking and behaving exactly the same on each platform and not needing to re-implement a UI abstraction layer for every supported platform. Apps looking the same on each platform can also be a downside as users expect a platform specific look and feel from apps. Using platforms' default UI widgets, such as webviews and maps, is supported but is still experimental on iOS [SIAIV, 2018]. Asynchronous non-view interoperability and adding a Flutter portion to otherwise native SDK app are supported on both platforms like with most tools.

Flutter apps are written in Dart, an object-oriented programming language created by Google. It was originally intended for creating web applications, but was later adopted for Flutter as its features, like supporting both interpreted and ahead-of-time compiled execution, fit Flutter's use case. With the ongoing legal case with Oracle suing Google for copyright infringement for using Java on Android [Al-Heeti, 2018], Google might also have wanted to use their own programming language for their next mobile SDK.

Building a UI with Flutter is based on nesting widgets in a tree structure. The app and its widgets are all written in Dart, so no separate markup or template language is used for

defining the UI. Styling the layout and even animations are also created with widgets. For example, padding is added by wrapping the desired widget with a Padding widget. Flutter has out of the box implementations for Material design and iOS styled widgets, but no automation to change which one is used on each platform.

As Flutter has really gained attention from developers only recently and Dart is not used widely outside of Google and the Flutter community, the amount of community created modules can be smaller when compared with some of the other tools. Flutter's documentation is comprehensive with API reference, examples and adaptation guides for developers familiar with other tools.

5. Comparing performance between tools

This thesis' main goal is finding out if apps created with current generation of cross-platform development tools are inferior to apps created with native SDK tools in their performance. To achieve this, we need to define ways on how we can compare apps created with different tools in a reliable and fair fashion. This chapter goes through how we define performance and how it manifests in the apps, how previous studies have done their measurements and how this study brings something new to the table by adapting a measuring method not previously used in this context. We'll also go over how the tests were ran in the comparison study.

Previous studies have used various ways to compare these kinds of tools and for our purposes we can roughly group them to three groups: performance, resource usage and others. Going quickly over the two groups we are not looking at in this study, the resource usage group contains metrics that can be measured either from the app installation file or from a running instance of the app. As its name implies, it contains metrics on how much system resources it needs, such as CPU or RAM usage of the app while running or the size of the installation file. The others group contains metrics and other assessments related more to the tools themselves rather than the finalized apps created with them, like the development process, code complexity, documentation and tooling. Both of these groups have some interesting metrics in them and it would be great to see a future study taking a look at the current generation of cross-platform apps from those perspectives, as neither Flutter or React Native have been part of those kinds of academic studies.

The performance group of metrics this study focuses on looks only at running instances of apps and how fast they can perform their tasks, like opening a new screen or calculating something. Several previous studies have compared apps with these types of metrics [Ohrt and Turau, 2012; Willocx *et al.*, 2015]. The types of tasks can be divided into four categories: state transitions, computation, I/O, accessing device features. The state transitioning category includes tasks such as moving between screens and launching the app. Computational tasks do some (heavy) calculations, like finding prime numbers or image manipulation. When comparing cross-platform tools it's important to note that for meaningful testing, the computation should be kept in the cross-platform portion of the app, and not be passed to the native code through the bridge. I/O tasks do read and write operations on the local disk and make network requests. One example of device feature accessing tasks is triggering the vibrator on the device, which was previously used by Corral *et al.* [2012] in their study. In the context of comparing cross-platform apps, if the different tools use similarly efficient native code to access the device features, most of the differences in performance should come from the efficiency of the bridging mechanism.

5.1. Logging-based measurement

Measuring the time in these types of performance has previously been done mostly by logging messages with a timestamp to a debug console or a file, the measured time being the difference between timestamps acquired before the task starts and after it finishes [Corral *et al.*, 2012; Willocx *et al.*, 2015]. This timing method should lead to millisecond level accuracy with most tools and some tools can even achieve nanosecond accuracy [Corral *et al.*, 2012]. It has some other great qualities on top of its accuracy, like ease of implementing it, being light enough work for the device to not affect the results and outputting text, which is easy to automatically transform into formats that analyzing software can take as input. Overall when thinking of timing methods for software performance, logging is probably the one that comes first in mind for many and fits many use cases quite well.

Despite its many good points, logging might not be suitable timing method for our testing. This might be easiest to see this through some examples. For our first example, let's say we want to start timing when a button is pressed. A native app would register a click listener to that button, in which it does the log call and then proceeds on the actual task. With cross-platform tools that use the native SDK for their UI, the tool abstracts away registering that native click listener and the event can potentially go through many layers of abstraction before reaching our code. So, we can't accurately start the timing when we want to, the moment the app receives the input from the OS, and instead have some arbitrary delay that varies between tools. As a second example of use cases where logging can be inaccurate, let's consider triggering the device vibrator from cross-platform code. On both Android and iOS, the native SDK method to trigger the vibration quickly passes an event to the OS, which will asynchronously then start the vibration, with no return values or callbacks. All of the cross-platform tools in our comparison have an abstraction that works in a similar fashion, just quickly pass an event to the bridge and let the cross-platform code move on. So, if we just naively take the time it takes for the tools abstracted function to trigger vibration to execute, we've most likely only measured how much time it takes to tell the bridge to send an event to the other side. We have no guarantee if the native side has already processed the event and even if it had, how much other processing had been done after that. Our last example is the app launch time measurement, and this issue affects even apps created with native SDKs. The time it takes to cold start an app is affected by the amount of code and resources that need to be loaded to the RAM before the operating system can even start the app, which means that we'd need to do our first logging call before we've had any chance of doing so. Looking at the app launch time from this perspective, the software we'd be testing would actually be the operating system and the task would be getting the app to the state we consider as fully launched.

Sometimes the issues mentioned above can be solved by modifying the tools' source code to do the log calls at the timing we really intend to, but with tools and SDKs that

aren't open-sourced this wouldn't be possible. Modifying the source code would also add lots of extra work and require knowledge of the tools' internal workings. Depending on the device we're testing on, we might get lucky and have access to operating system logs that tell us when our app is being launched. In many cases using the log timing should be accurate enough when comparing apps created with the same tool, but the code that we don't control creates too many unknown factors for it to be a reliable timing method when comparing apps created with different tools.

5.2. Video-based measurement

To overcome the issues with logging-based measurements, an alternative was needed. Inspired by video game input lag tests conducted with high-speed video cameras [Soomro, 2015], a video-based measuring method was adopted. With this method, a video of the device screen and inputs being given to the device is recorded, and the result is based on the number of frames it takes from the input being given to the task being finished. One exception in this study would be the vibration triggering test, where timing ended when the sound of the vibration started in the video. The precision of this type of timing would be tied to the frame rate of the video and thus lose to logging in this regard, but the conditions for starting and ending the timing would be exactly the same for all kinds of apps and the results can be directly compared with no issues. As long as all the frames shown on the device are recorded, the accuracy should be good enough to measure how a user would perceive the apps' performance.

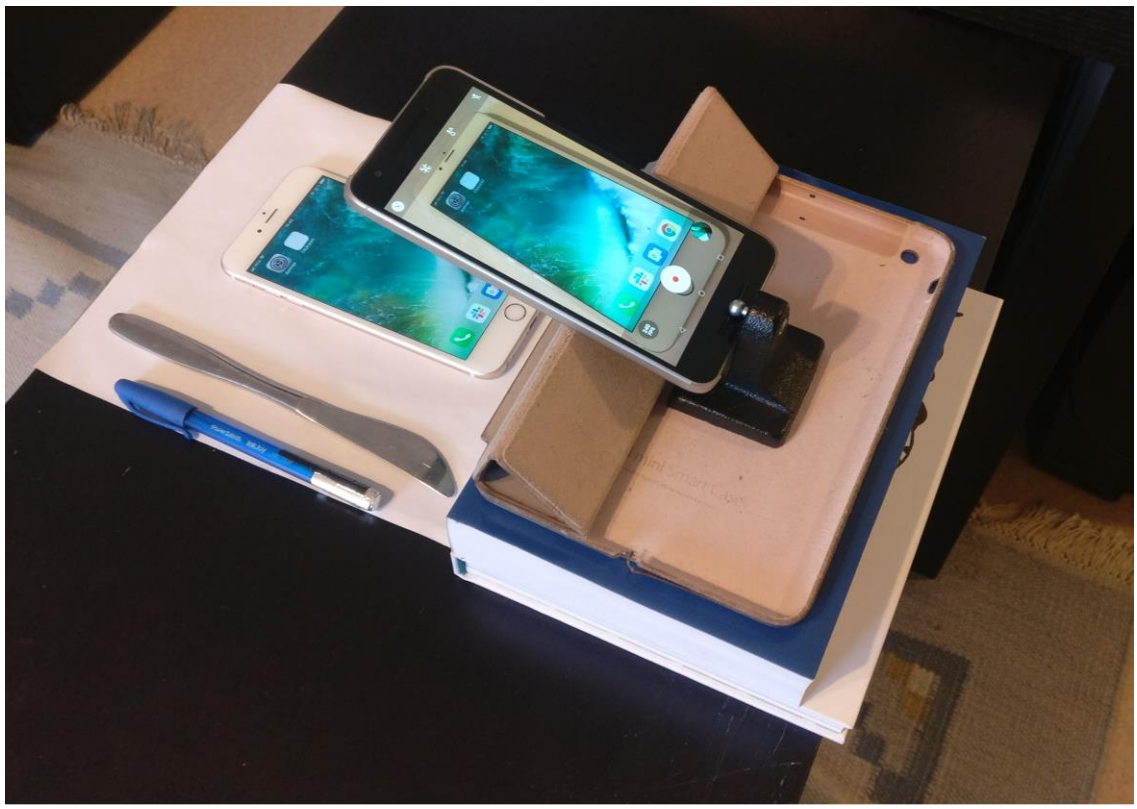

Figure 1. The high-speed camera video recording setup with a Nexus 6P acting as a camera.

The initial batch of test runs were conducted by directly applying Soomro's [2015] method of using a high-speed video camera. Figure 1 shows how the recording was setup with the camera being rigged close to the device being tested in an angle that left clearance for a stylus to touch the screen. After initial analysis of the videos however it became clear that this recording setup had too many flaws for the results to be considered accurate. The camera used for recording was the Nexus 6P smartphone from 2015, which has a 240-fps slow-motion video recording mode. The camera often produced grainy videos where it was hard to see exact timings and it was slow to adjust the colors when the screen faded quickly from one color to another. This created situations where it was hard to judge e.g. a fade from blue to white ending when even the paper behind the phone glowed with blue color for a while in the recording. The IPS panel on the iPhone 6S Plus used as a test device was problematic as changes that were supposed to be instantaneous happened as fades on the video, which added extra complexity to definitions on when a task is considered to be finished. It was also hard to determine the exact frames where inputs were given when a regular stylus pen or a finger was used to touch the screen, as the motion of touching the device screen with them moved away from the camera and the two surfaces making contact can't be seen on the video. To make the two surfaces making contact visible on the video, a custom-made stylus made by wrapping aluminum foil to tip of a pen was used as a proxy. The stylus couldn't trigger inputs by itself and it needed a finger or a metal object to touch the foil for the screen to register inputs from it.

As shown in Figure 2, by placing the stylus on the screen and then touching the foil with a knife, inputs could be sent to the device with the two surfaces making contact (the stylus and the knife) being caught in the video. While this worked great in many recordings, there were still hard-to-judge situations when the stylus wasn't in the center of the camera's focus, or the knife used to touch the stylus was tilted.

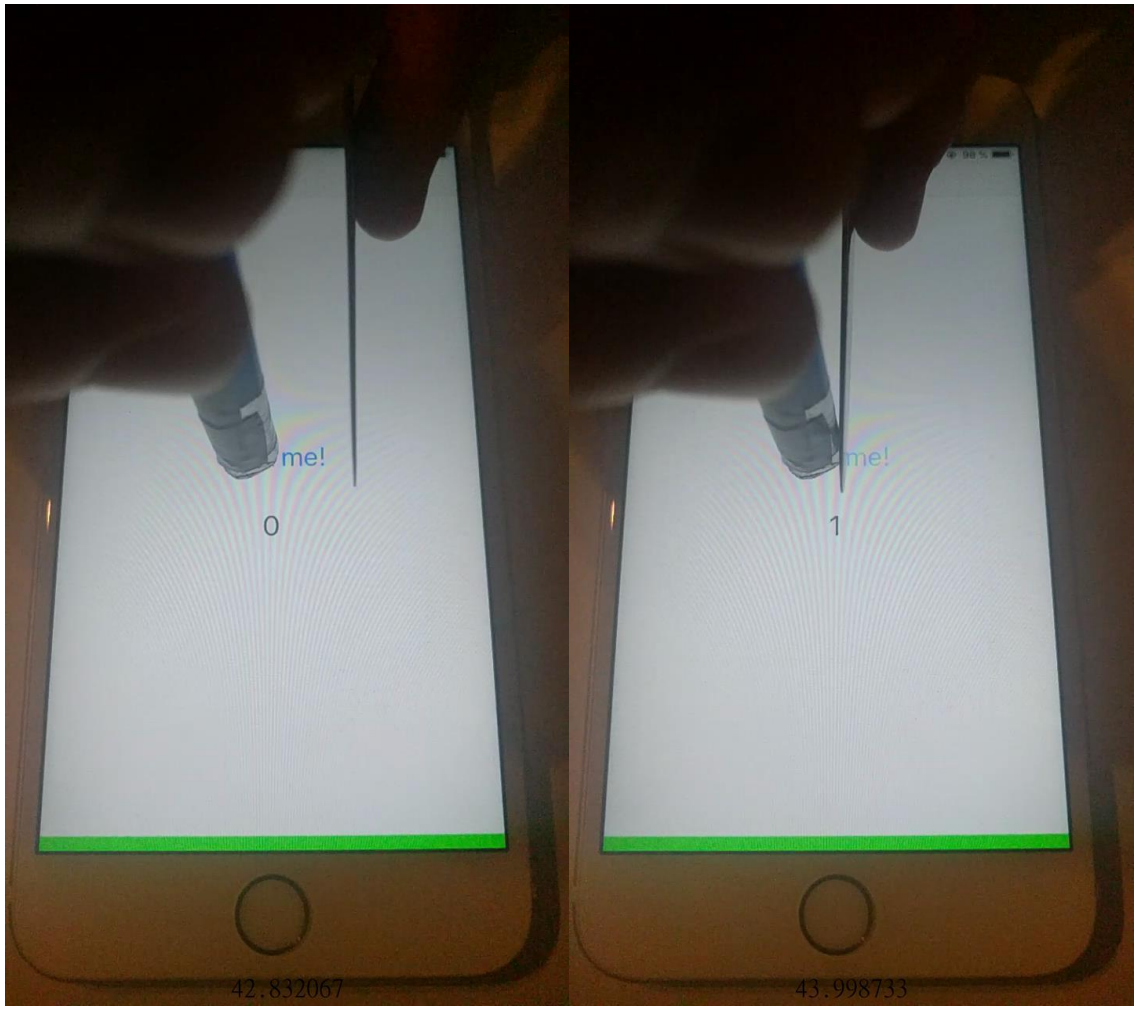

Figure 2. Two frames from a high-speed video camera recording demonstrating how the custom stylus was used.

The second batch of test runs was done with screen recording software while an on-screen marker for inputs was enabled on the device. This got rid of all the image quality issues that the camera recordings had and there was no ambiguity on which frame was the one that the input was given. In a way it also made the timing closer to what we're after by moving the starting time from the moment of input being given to the screen to the moment that the operating system had received it, removing the time it takes for the hardware to register the input. This makes the results comparable even between devices with different types of screens. The biggest issue with screen recording on Android and iOS devices is that all available software for it allow only variable frame rate recording. These types of recordings save space by not saving duplicate frames in the video, and

instead rely on metadata to tell the video player software on when to show the next frame. This makes analyzing the video frame by frame more complicated, as advancing the video by a frame doesn't advance the time by a constant, like it would do in a constant frame rate video (in smartphone use case, recording all of the frames of their 60Hz screens would generate a 60-fps video where every frame is shown for 16,666… milliseconds). Using screen recording software also creates extra work for the device, which can affect the results. This wasn't seen as a problem as the recording should affect all the apps with roughly the same impact, and the video camera recordings were used to check that the results were similar between the two recording methods.

On top of the four types of tasks (state transitions, computation, I/O, device feature access) done in performance testing, the video-based measurement also enables testing animation performance. It doesn't fit the previously given definition of performance metric as animations aren't always tasks that have well defined start and end times, and faster isn't always better. Rather, animation performance would be how smoothly the app can run its animations, which can't really be measured through logs at all. With video, animation performance could be measured by going through the video frame by frame and counting the amount of times there's an unexpected duplicated frame. Adding directly comparable animations for apps created with 6 different tools per platform would be quite a difficult task, so no animation performance testing was done in this study. Something similar however was done with scrolling through a list of items and counting the amount of dropped frames during the scroll. The previously mentioned issues with access to only variable frame-rate recordings complicates this type of testing, as the frames stored in the video file aren't the single source of truth if the playback will have duplicated frames or not like they would be in a constant frame rate video. So, the amount of dropped frames during a scroll was counted by first checking the duplicated frames, and then adding the number of frames that were shown for over 33,333… milliseconds during the scroll. If a frame was shown for over 50 milliseconds, it was added twice and so on.

5.3. Testing setup

The video-based performance method was put to a use in this study by running tests on apps created with tools introduced in Chapter 4. The tests were run on a OnePlus 3 running Android Oreo 8.0 and iPhone 6S Plus running iOS version 12.1.2. During the testing the phones were connected to a computer with USB cable and the reported battery charge level was over 95% while tests were run. The phones didn't have SIM cards installed, were connected to a 5GHz Wi-Fi with a good connection quality and had automatic updates and app synchronizations turned off. Before each test run the app under testing was installed, then launched once to make sure any first launch setup was done, the device was rebooted, and then the device was left idle for at least a minute to make

sure that most post-launch procedures had finished. The reinstallations were necessary as some of the cross-platform tools automatically cache images loaded from the Internet and the reinstallation was the most surefire way to delete the cache between tests. The reboots were done to make sure that the phone is in a similar state regarding background processes and such for each test.

Tests that fetched content from the Internet connected to an Apache HTTP server that ran on a Raspberry Pi 3 in the same network, connected with an ethernet cable. The Pi didn't do any other work during testing and the fact that the server was running smoothly was confirmed before each test recording session.

Making touch inputs visible in the recordings on Android was as easy as enabling the "Show touches" option in the developer options menu. There is also a separate "Pointer location" option that shows inputs with some extra data, but that option occasionally was a frame behind the marker shown by "Show touches" option, so it wasn't used. iOS doesn't have a dedicated option to show inputs, but the Assistive Touch accessibility option was used as a replacement. Assistive Touch can be used to replay macro inputs with a single touch, and by recording a quick tap as a macro it can effectively be used to show touches. Assistive Touch always shows a marker on the screen, but when the macro activates it flashes, so the time the flash starts was used as indication that the input was given. For scrolling tests, a swipe gesture macro was used. On Android there is no built-in macro system and the "Show touches" marker doesn't appear for inputs given through USB, so the swipes were done manually.

Most of the video recordings on Android were done with the screen recording tool that comes as a part of ADB (Android Debug Bridge). This method stored the videos on the device and has an option to embed frame numbers and timestamps for each frame to the video. However, it doesn't support recording audio, which was needed in the vibration triggering tests as the sound of the vibration motor was used to determine when the timing should end. Because of this, the screen recording app AZ Screen Recorder (https://play.google.com/store/apps/dev?id=4946092157052757127) was used to record the vibration tests. Using scrcpy (https://github.com/Genymobile/scrcpy) was also considered as it would've streamed the video to a computer and thus not write to disk during testing. The ADB screen recording was chosen over other tools as it was the only tool that didn't make the 60-fps animation test at Test UFO (https://testufo.com) warn about not being able to achieve steady 60-fps. On iOS the only option without jailbreaking the device was to record the screen using QuickTime on a Mac with excellent quality setting. iOS disables the device's vibrator while the microphone is in use, so an external microphone was needed for the vibration tests. The microphone on the MacBook that ran QuickTime was used as it didn't need any extra setup. The phone was placed on top of the MacBook's keyboard during vibration triggering test so the distance between vibrating device and microphone was same for all recordings.

6. Benchmarking app

To make performance comparisons between apps created with native SDKs and different cross-platform tools, apps created with all of them were needed. To ensure that the comparisons between apps is meaningful and fair, specification for a benchmarking app was defined and then implemented with each tool. The benchmarking app was designed to be relatively simple to create with each tool and allow easily testing different aspects of them. That's why a design with a simple landing page that had buttons that navigated to different screens was chosen. Each screen contained everything needed for a single type of test, like a counter that incremented when a button was pressed or a large list of items. It might not have resembled a typical real-world app in its design but kept the tests isolated, so they could have been easily changed without affecting other tests. The versions of the tools and other packages used were the latest official releases on December 20th, 2018.

The benchmarking app specification given in next sections contains rough UI descriptions, sample screenshots from the finalized apps, feature specifications, general implementation guidelines and some specific implementation requirements that were followed. Implementation requirements were specified in places where the same feature or UI design could have been implemented in multiple ways with different kinds of performance implications. For example, some parts of app UI were specified to be static and some dynamically created.

All the implemented apps are open-sourced and can be found at https://github.com/jarkkos-crossplatform-comparison-thesis.

6.1. General implementation guidelines

The general implementation guidelines were the following:

- The UI should be implemented with official resources and officially endorsed third-party resources.
- For non-UI purposes all official addon packages and free & open-source libraries are allowed.
- Custom made platform specific code should be kept to minimum.
- Follow each tool's coding style and best practices, don't try to optimize for testing.

These rules were given to make the apps' implementations push the cross-platform capabilities of the tools' official packages. The definition of official resource was that it is distributed by the same organization as the tool itself and it's not labeled as experimental, like the Android support library by Google. If the official documentation suggests a specific third-party library or module for doing certain tasks, it was considered to be an officially endorsed resource. The clause for officially endorsed third-party

resources was added because neither ReactJS or React Native had built-in solutions for moving between screens, but their documentations suggested third-party solutions for the task [RJSRouting, 2018; NBS, 2018]. The second rule was given to speed up development and to bring the implementations bit closer to what they would be in average apps.

Because Cordova isn't a tool for creating mobile apps but a tool for packaging a web apps as mobile apps, some extra considerations needed to be made. The web app to be packaged with Cordova was created with ReactJS so it could share non-UI code with the React Native app. The general app implementation guidelines were then applied to ReactJS.

All screens of the app had a bar with the title of the app or the screen. It will be referenced as the "top bar" as the platform specific terms "navigation bar" and "toolbar" have different meanings on both platforms. In the test screens, the top bar contained a button to go back to the landing screen.

All the buttons were configured to do their actions on touch down events, or when a finger initially touches the screen. The norm on both platforms and by default followed by all the tools is to do the action on touch up, or when the finger is lifted from the button. The default behavior allows canceling the button click by moving the finger away from the button before lifting it, but the touch down event is easier to pick up from a video so it was used instead. Most tools allowed using touch down events out of the box, while with React Native and Flutter the source code of the default buttons needed to be copied and modified to create widgets with touch down event listening support.

Another change from the default behavior for testing purposes was to disable all screen transition animations and delays. Different tools could have animations with different timings, which would make the comparison be about some animations timing parameters defined by the tool instead of how fast the apps can navigate between the screens at their fastest. All tools allowed disabling animations when moving to a new screen, a feature presumably added for state restoration and deep linking purposes, but moving to a previous screen with default controls was often more difficult to implement. Even the native iOS SDK didn't support disabling animation of the default back button. With most tools this could be worked around by using a custom button for moving back, the only exception being Flutter which allowed disabling the animation but still seemed to delay the screen transition arbitrarily.

6.2. Landing screen

The landing screen was the first screen that is shown when the app was launched. It worked as a hub for accessing all the other screens of the app. While the landing screen didn't have any test specific functionality in it, it was used to test app launch time and as part of tests that involved moving from one screen to other.

The design of the landing screen layout was minimalistic. At the top there were two images in both corners of the screen, and between the images were three lines of text describing the app and its version. Below the images and title texts were a set of buttons, one below the other center aligned. These buttons moved the app to screens that contained the tests, and the buttons contained a text that describes the screen it opens. A red bar was added to the bottom of the screen to help determining when the whole screen had been rendered in the video camera recordings where the high frame rate recording captured partially updated screens. Figures 3 and 4 on the next pages show how the landing page looked in each app on both platforms.

Because the landing screen's initial rendering time affects the tests it was used in, its implementation was dictated strictly to make the implementations comparable. The layout had to be completely static with everything hardcoded. With tools that use separate code and layout files, the button click listener bindings were done in code. The code that the button click listeners execute to move to the next screen knew what to do without doing any sort of lookups for which screen to show or what kind of configurations to pass to it. These implementation requirements were given to make sure that all the implementations take the same approach of prioritizing execution speed over other things like configurability of the buttons and the screens they lead to.

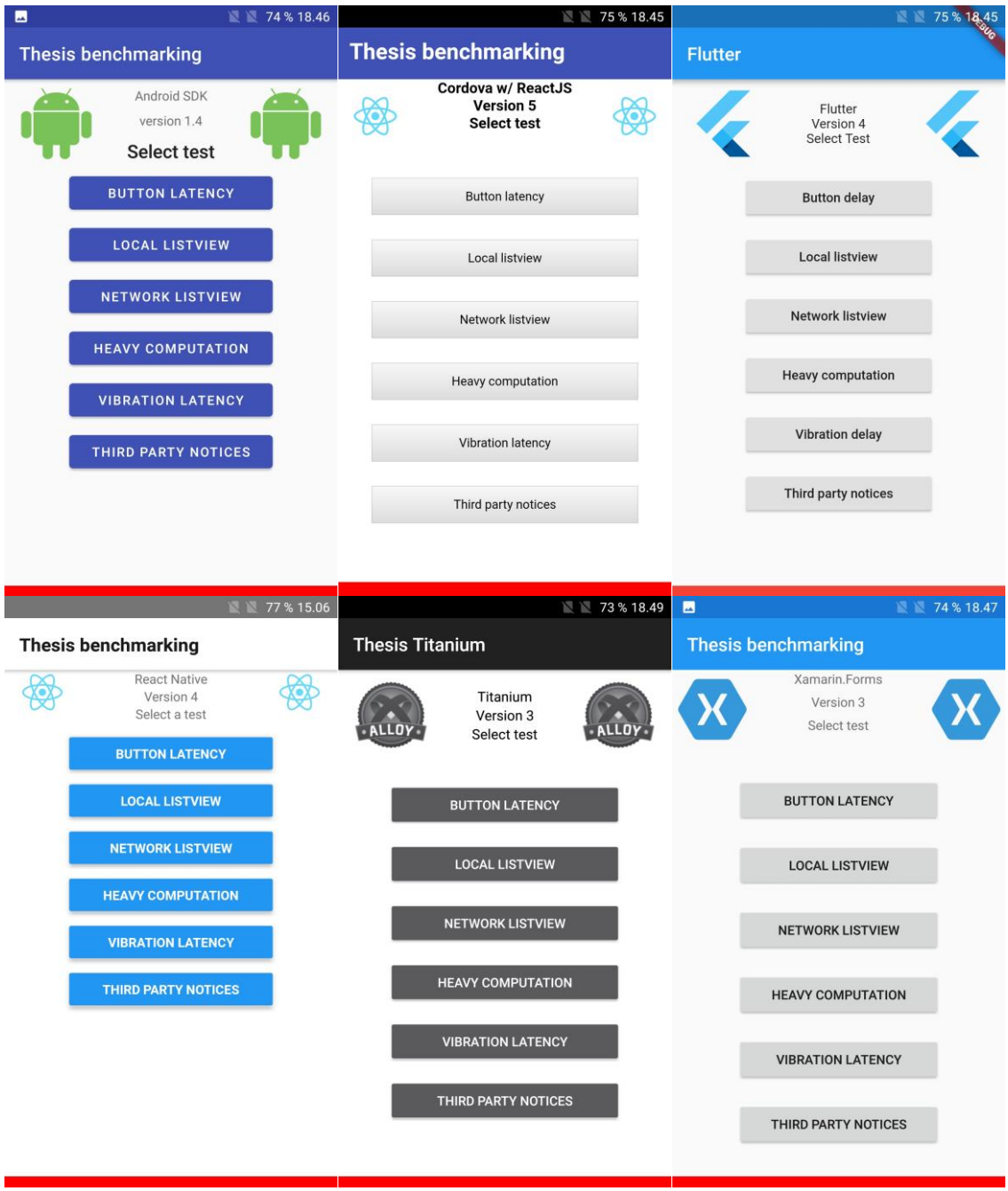

Figure 3. The landing screen in all 6 Android apps.

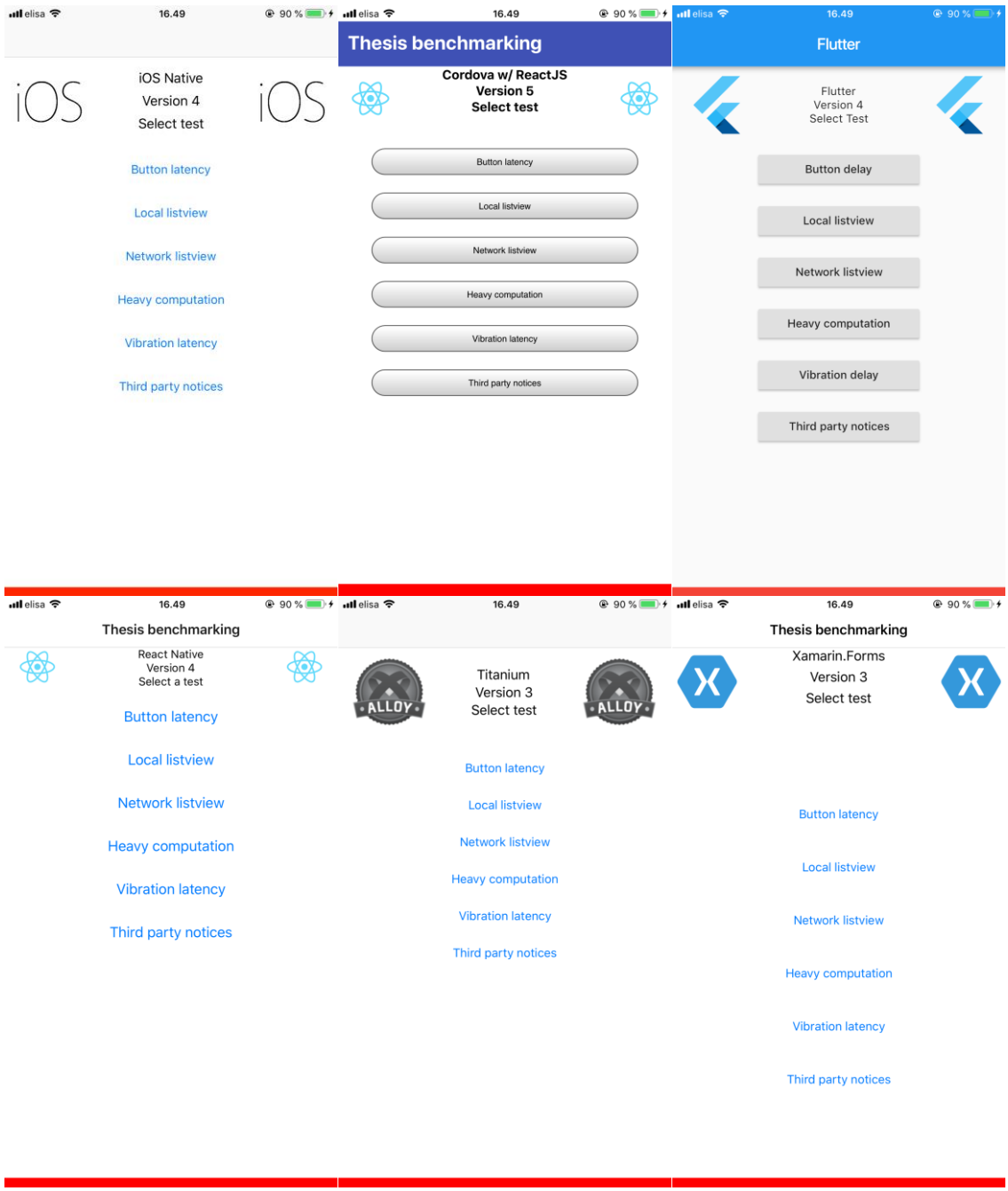

Figure 4. The landing screen in all 6 iOS apps.

6.3. Button reaction delay screen

The button reaction delay screen contained a top bar, a button in the middle and a line of text below the button that showed how many times the button has been pressed. Figure 5 shows how the screen looked in native apps on both platforms. The apps were implemented so that the layout is static, and the button click listener is bound in code. The amount of times the button has been pressed was kept in an integer variable with hardcoded initial value of zero. This variable was stored in an appropriate place to store simple pieces of UI state for the tool and solutions used with complex state (e.g. view model, redux store) were avoided in the implementations. Clicking the button incremented the counter variable by one and made the UI to update with the updated value. A green bar was added to the bottom screen for the video camera recordings, similar to the red bar on the landing screen.

As none of the cross-platform tools supported using touch down events with the default back button, and with many the animation from the default back button couldn't be disabled, a separate button for back navigation was added to this screen for back navigation speed testing.

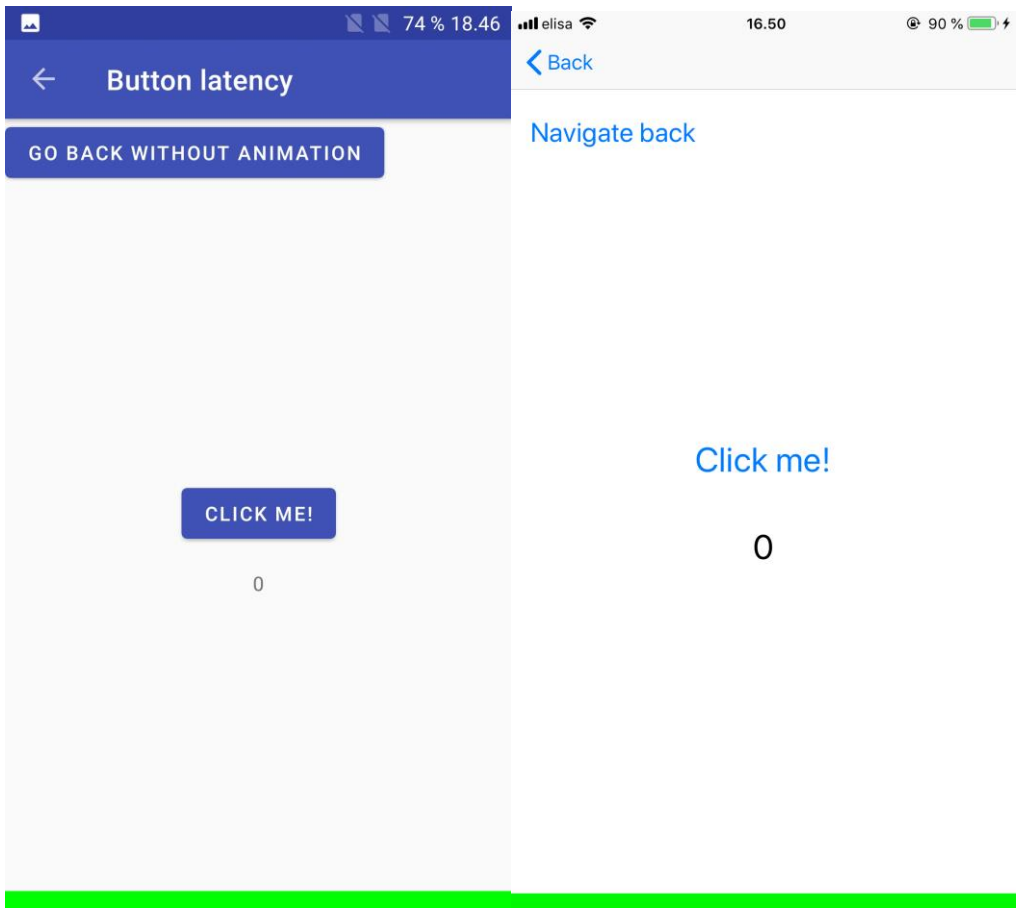

Figure 5. The button reaction delay screen in the native Android app and the native iOS app.

6.4. Large list of items screens

These screens simulate a common task for a mobile app, loading a list of items and showing them to the user. The item data was stored to a json file, with each item containing information of a jpg image's URI and a sentence of text. Images used were from the Caltech 101 dataset [Fei-Fei *et al.*, 2004] and the texts were generated at lipsum.com. As shown in Figure 6, the apps displayed this data in a vertical list where each item's image was shown in the left side of the screen and the text next to the image. The height that a single item used was adjusted so that on both platforms the same number of items was shown in all of the apps (this ended up being 7 items on Android and 8 + 9th peeking from the bottom on iOS).

To test the list scrolling performance of the tools, the list had 300 items to create long enough list that a single swipe couldn't scroll through it. Most production quality apps would implement paging to loading such a large list of items either with infinite scrolling and loading more items on demand or by splitting the list to different pages. The approach of loading data for all the list items at once and letting the tool handle performance optimizations was used as it was easy to implement with all of the tools and could make the performance differences between tools more apparent than a more typical number of items.

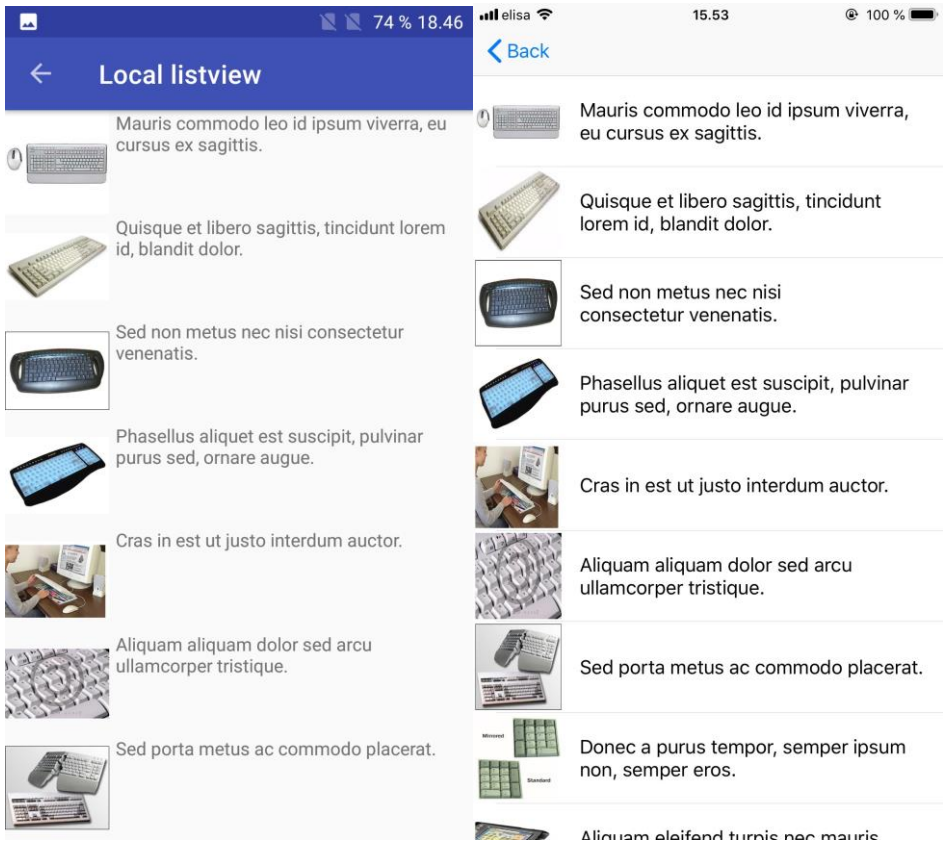

Figure 6. The item listing screen in the native Android app and native iOS app.

Each app had two versions of this screen: one that read the data from disk and one that fetched it from the Internet. This enabled testing how the tools could handle reading data from these two common data sources and if the choice of data source has any effect on the list scrolling performance. While not an actual requirement or a focusing point of the study, all apps had both versions of the screen use the same layouts and display logic, only swapping the implementation of data loaders. This was done to reduce duplicate code, to see how the APIs of the tools could handle loading images from different sources and to test how simple dependency injection could be done with each tool. All the tools could handle this type of implementation fine, with the only exception being that the data loader's implementation couldn't be injected to the screen in the native Android implementation due to how instantiating Fragments works. To get around this, a flag describing which implementation the Fragment should instantiate by itself was injected instead.

6.5. Heavy computation screen

Like the button reaction delay screen, the heavy computation screen had a top bar and a button, but the content below the button changed a bit more depending on the state. Initially there was a text prompting the user to click the button. When the button was pressed, it triggered the computation to start in the background, the text changed to "Computing" and a spinner or similar animated UI widget was shown. After the computation was done, the spinner was hidden, and text was changed to "Done". Figure 7 shows how the screen looked in native apps during the computation with the spinner visible.

The computation task chosen was finding all the prime numbers below three million and returning them as a list. This task was chosen as it was easy to implement optimally on all platforms with similar syntax. A more realistic computational task for a mobile app could've been some image manipulation task like applying a filter to an image but creating optimal and comparable solutions for each platform would've been difficult.

Doing the computation in the background meant that it had be done asynchronously in another thread or similar construct in a way that it didn't block UI updates or user input. With this kind of implementation, the computation doesn't disrupt the spinner animation and the back buttons still works like users would expect in a normal app. With tools that supported it, the computation was implemented in a cancelable manner and was cancelled either when the button that started the computation was repressed or back button was pressed. The apps weren't allowed to explicitly initialize the mechanism for doing the computation in a background thread before the button was pressed.

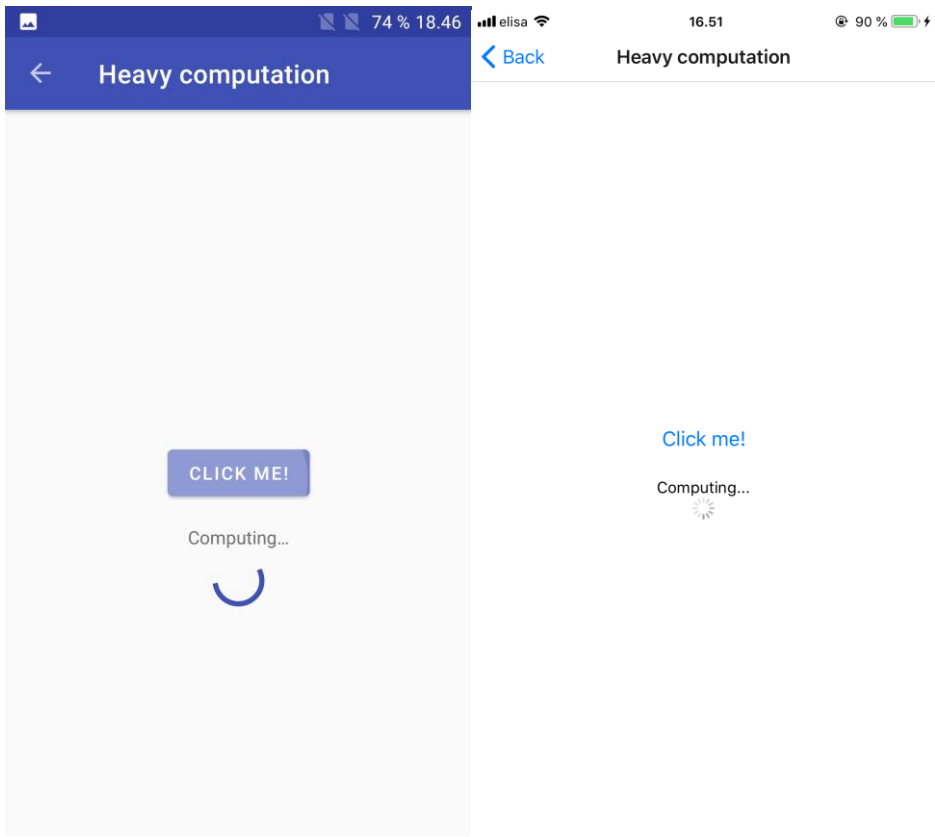

Figure 7. The heavy computation screen in the native Android app and the native iOS app during the computation.

6.6. Vibration screen

The vibration screen had a static layout with just the top bar and a button in the middle of the screen, like the ones in Figure 7 but with no additional widgets below the button. Pressing the button makes the device vibrate for a moment. Triggering the vibration to start had to be implemented by fetching a reference to the bridging mechanism used to control the device vibrator and calling the appropriate method. Fetching the reference to the bridging mechanism before the button got pressed wasn't allowed to make sure that it didn't cause any sort of vibration specific initialization to happen before the test timing started.

7. Results

The results for each kind of test are given in a section specific for each category of tests, with each section containing the results in table(s) and text pointing out the most noteworthy results and explaining any oddities. In the result tables, React Native is abbreviated as "RN". When multiple tools have the same result in a test, the table has them grouped in an alphabetical order.

The results presented in this chapter are from the screen recording software tests, with the only exception being the Section 7.1. where the results between the two tested video recording methods are compared. Tables in that section have the recording method mentioned with the type of test they give the results for, but in later sections the recording method is omitted as they all have only screen recording software results.

Due to the large amount of manual work with recording the videos and going through them to find the frames that start and end the timing, the tests were run only three times for each app. The main comparisons were done by choosing the fastest one of the three runs. There were two reasons for selecting the fastest one over the second fastest and slowest times. The fastest times are good estimations of how the apps can perform at their absolute fastest, while with the sample size of three we can't really be sure what an average result would be in the long run and aiming to find the slowest results that aren't caused by the device randomly doing something in the background isn't really meaningful. The second reason was that the fastest results of video camera and screen recording software tests aligned with each other better than the other two. Using the mean time of the three samples wasn't even considered due to the small sample size and problems that having times that aren't approximately multiples of 16,666… milliseconds could cause when converting them to approximated amount of 60-fps video frames.

7.1. Comparing high-speed camera and screen recording software results

As explained previously in Chapter 5, a lot of videos were already recorded with a high-speed video camera before the switch to screen recording software was decided. As those videos already existed, the ones that didn't have too much issues with image quality or clarity on timing were compared with the screen recording videos to validate that the screen recording going on during testing didn't affect the results significantly. The comparison wasn't done on the network list and vibration tests, as doing two list tests seemed unnecessary and the vibration test result accuracy was questionable anyway as will be discussed later in Section 7.6. The local list loading test was chosen over the network version, as both list loading and screen recording on Android accessing the disk at the same time could potentially make the performance hit of using screen recording software more apparent. Only the apps' relative performance to each other was compared, as the absolute times have different variables affecting them with each method (e.g.

registering the stylus touch from the screen and passing it as an input event to the app) and aren't directly comparable between the two recording methods.

Table 1 contains the fastest time comparisons for Android. Most of the differences are apps with close timings switching places with each other, which can either be because of using screen recording software affects the results slightly or by random with the small sample size of recordings. Cordova's ranking relative to Flutter and React Native changing between the testing methods in some tests can probably be explained by the two larger apps being more affected by the ongoing disk writes, but the largest app Xamarin didn't seem to mind the writes as much. While the absolute times can't really be compared, React Native's slow touch registering in camera tests and the performance hit that Flutter takes from the screen recording in computation test are worth mentioning even though they didn't cause any changes in placements.

software launch time	tool	camera launch time	tool
0:00,451	Native	0:00,450	Native
0:00,920	Cordova	0:00,892	Flutter
0:01,050	Flutter	0:01,029	RN
0:01,773	RN	0:01,054	Cordova
0:02,224	Titanium	0:01,842	Titanium
0:02,280	Xamarin	0:02,325	Xamarin

software new screen	tool	camera new screen	tool
0:00,016	Native	0:00,092	Flutter
0:00,038	Flutter		Native
0:00,050	Cordova	0:00,150	Cordova
0:00,117	Xamarin	0:00,192	Xamarin
0:00,185	Titanium	0:00,258	Titanium
0:00,267	RN	0:00,300	RN

software previous screen	tool	camera previous screen	tool
0:00,031	Cordova	0:00,096	Native
0:00,048	Native	0:00,121	Xamarin
0:00,066	Xamarin	0:00,146	Cordova
0:00,101	Titanium	0:00,183	Titanium
0:00,133	RN	0:00,304	RN
0:00,367	Flutter	0:00,504	Flutter

software button press	tool	camera button press	tool
0:00,000	Flutter	0:00,067	Native
	Native		Xamarin
	Titanium	0:00,071	Titanium
	Xamarin	0:00,079	Flutter
0:00,015	Cordova	0:00,083	Cordova
0:00,065	RN	0:00,217	RN

software local list	tool	camera local list	tool
0:00,104	Flutter	0:00,200	Flutter
0:00,167	Native	0:00,225	Native
0:00,286	Titanium	0:00,367	Titanium
0:00,402	Cordova	0:00,600	Cordova
0:00,454	RN	0:00,613	RN
0:01,138	Xamarin	0:01,242	Xamarin

software computation	tool	camera computation	tool
0:00,869	Native	0:00,921	Native
0:01,355	Titanium	0:01,546	Titanium
0:01,595	Cordova	0:01,692	Cordova
0:14,724	Xamarin	0:15,321	Xamarin
0:57,986	RN	0:49,608	RN
2:13,742	Flutter	1:55,533	Flutter

Table 1. Comparing fastest screen recording software and high-speed camera results on Android, sorted by the amount of time.

Table 2 has the same comparisons for iOS. Here we have some rankings switching places, but the tables here make the differences seem a bit worse than they actually were. The camera results have finer accuracy (4.166… milliseconds) compared to the screen recording software results (16,666… milliseconds), which caused the screen recording results to round up to exact same times more often than the camera results. These grouped results are given alphabetically in these tables, and if we look at the camera results for apps that had same results in screen recording tests, we see that the differences are often close enough that they could've been rounded to be the same if they were measured with screen recording. This explains most of the placement switches in new screen, button press, local list and computation tests in Table 2 and button press test placement switches in Table 1. The Cordova-Titanium switch in Table 2's computation results is most likely due to random chance in this small dataset, as a 30-millisecond difference in a test that runs for roughly 27 seconds isn't significant. This leaves the launch time and previous screen test as the only tests with unexplained switches in their results in Table 2. Some of them might come from inaccuracies in timing, as the slowly responding IPS panel pixels made especially the launch time and screen changing tests hard to judge and those exact tests were the final deciding factors for the switch to screen recording software.

software launch time	tool	camera launch time	tool
0:00,683	Titanium	0:00,638	RN
0:00,700	RN	0:00,654	Native
0:00,783	Native	0:00,708	Titanium
0:00,967	Xamarin	0:00,775	Xamarin
0:01,250	Flutter	0:01,158	Flutter
0:01,330	Cordova	0:01,271	Cordova

software new screen	tool	camera new screen	tool
0:00,050	Flutter	0:00,104	Native
	Native	0:00,113	Titanium
	Titanium	0:00,125	Flutter
0:00,067	Cordova	0:00,133	Cordova
	Xamarin	0:00,146	Xamarin
0:00,083	RN	0:00,150	RN

software previous screen	tool	camera previous screen	tool
0:00,017	Native	0:00,067	Titanium
0:00,033	Cordova	0:00,071	Native
	Titanium	0:00,100	Cordova
	Xamarin	0:00,125	RN
0:00,067	RN	0:00,158	Xamarin
0:00,417	Flutter	0:00,492	Flutter

software button press	tool	camera button press	tool
0:00,000	Native	0:00,046	Native
0:00,017	Cordova	0:00,058	Titanium
	Flutter	0:00,067	Flutter
	RN		Xamarin
	Titanium	0:00,071	Cordova
	Xamarin	0:00,075	RN

software local list	tool	camera local list	tool
0:00,133	Flutter	0:00,171	Flutter
0:00,217	Xamarin	0:00,258	Xamarin
0:00,267	Native	0:00,313	RN
	RN	0:00,333	Native
0:00,600	Cordova	0:00,633	Cordova
0:00,700	Titanium	0:00,700	Titanium

software computation	tool	camera computation	tool
0:00,900	Native	0:00,921	Xamarin
	Xamarin	0:00,929	Native
0:01,284	Flutter	0:01,208	Flutter
0:27,470	Cordova	0:26,787	Titanium
0:27,504	Titanium	0:26,800	Cordova
0:27,821	RN	0:27,146	RN

Table 2. Comparing fastest screen recording software and high-speed camera results on iOS, sorted by the amount of time.

7.2. App launch times

Android

approx. frames	tool	time
27	Native	0:00,451
55	Cordova	0:00,920
62	Flutter	0:01,050
105	RN	0:01,773
131	Titanium	0:02,224
135	Xamarin	0:02,280

iOS

approx. frames	tool	time
41	Titanium	0:00,683
42	RN	0:00,700
47	Native	0:00,783
57	Xamarin	0:00,967
74	Flutter	0:01,250
79	Cordova	0:01,330

Table 3. Fastest app launch times on Android and iOS, sorted by the approximated number of frames. Timing is from the input to open the app being given to app's first screen being completely drawn with animations finished. As the app icon reacts to touches on touch up event, the frame of input being given was when the touch ended.

For the rest of this chapter, the results given in the tables are from screen recording software tests, and results from camera tests are separately given and pointed out only when necessary. In Table 3 we have some interesting results on the iOS side, with two cross-platform apps beating the native implementation in launch time. The native iOS app was implemented by using a single big Storyboard to define all the screen layouts, so it's possible that the native app's launch time was slowed down by the need to parse a file that contains layout information for more screens than just the landing screen.

On iOS the times were somewhat close together with result being either bit under or over one second, but on Android the times were quite scattered. The top 3 iOS apps had their screens drawn before the default animation of the screen expanding from the app icon had finished, and there doesn't seem to be any information available on what controls that animation's speed, so the timing for them was ended when that animation ended to make sure that the timing works the same for all of the apps.

While on iOS one couldn't probably tell from launch times only if the app is using the native SDKs or not, on Android the penalty was easily noticeable. The absolute slowest launch time in the dataset on iOS was Cordova getting 1,334s twice, which is practically the same as the fastest time it got. On Android Titanium and Xamarin took over three seconds in some test runs, with Xamarin's slowest time being 3,413s (3,579s in camera tests).

As mentioned earlier, on Android Cordova's relative performance against Flutter and React Native was worse in the video camera-based testing but in screen recording tests it was ahead of them. Whether it's due to small sample size or the screen recordings disk writes affecting larger apps, it's hard to say but should be considered when analyzing the results.

7.3. Moving between screens

Two separate types of screen transitioning tests were conducted: the first type moved into a new screen with a simple design and then back to the previous screen, and the other type moved to a new screen that had to load a list of items and dynamically create the layout before it could be shown. Table 4 has results of the former type of test and Table 5 of the latter.

At first glance, the most interesting result in Table 4's simple screen transition times is Flutter's slow transitions to previous screen. This was due to the Flutter's Navigator API not having an option to completely disable the animation and the time reserved for playing it, as giving it an empty animation didn't change the time it took to go back to previous screen. The Navigator API also didn't offer any easy way around the issue, so we're left with a result that tells us more about Flutter's API than its performance.

On actual as-fast-as-possible screen transition results, we have the native implementations being in the fastest groups in most of them, which isn't too surprising. On Android Cordova still managed to beat the native app though. Flutter also redeemed itself with the top 2 ranks on both platforms when moving to a new screen. React Native had consistently the slowest transition times. The React Navigation library that was used (and recommended by React Native documentation [NBS, 2018]) didn't have an explicit option to disable animations, but one could configure an animation that caused an instant transition to happen.

Android
open new screen

approx. frames	tool	time
1	Native	0:00,016
3	Flutter	0:00,038
	Cordova	0:00,050
7	Xamarin	0:00,117
11	Titanium	0:00,185
16	RN	0:00,267

Android
open previous screen

approx. frames	tool	time
2	Cordova	0:00,031
3	Native	0:00,048
4	Xamarin	0:00,066
6	Titanium	0:00,101
8	RN	0:00,133
22	Flutter	0:00,367

iOS
open new screen

approx. frames	tool	time
3	Flutter	0:00,050
	Native	0:00,050
	Titanium	0:00,050
4	Cordova	0:00,067
	Xamarin	0:00,067
5	RN	0:00,083

iOS
open previous screen

approx. frames	tool	time
1	Native	0:00,017
2	Cordova	0:00,033
	Titanium	0:00,033
	Xamarin	0:00,033
4	RN	0:00,067
25	Flutter	0:00,417

Table 4. Fastest screen transitions between simple screens on Android and iOS, sorted by the approximated number of frames. Timing is from input to open the new screen being given to the destination screen being completely drawn.

Android
local screen transition

approx. frames	tool	time
3	Flutter	0:00,035
4	Native	0:00,066
14	RN	0:00,236
15	Xamarin	0:00,249
17	Titanium	0:00,286
21	Cordova	0:00,352

Android
local full render

approx. frames	tool	time
7	Flutter	0:00,104
10	Native	0:00,167
17	Titanium	0:00,286
24	Cordova	0:00,402
27	RN	0:00,454
67	Xamarin	0:01,138

Android
network screen transition

approx. frames	tool	time
2	Cordova	0:00,034
3	Flutter	0:00,049
4	Native	0:00,066
10	RN	0:00,169
15	Xamarin	0:00,249
19	Titanium	0:00,317

Android
network full render

approx. frames	tool	time
10	Flutter	0:00,165
11	Native	0:00,184
23	Titanium	0:00,382
29	Cordova	0:00,485
46	RN	0:00,768
124	Xamarin	0:02,092

iOS
local screen transition

approx. frames	tool	time
4	Flutter	0:00,067
	Native	0:00,067
5	Xamarin	0:00,083
16	Cordova	0:00,267
	RN	0:00,267
42	Titanium	0:00,700

iOS
local full render

approx. frames	tool	time
8	Flutter	0:00,133
13	Xamarin	0:00,217
16	Native	0:00,267
	RN	0:00,267
36	Cordova	0:00,600
42	Titanium	0:00,700

iOS
network screen transition

approx. frames	tool	time
2	Cordova	0:00,033
4	Flutter	0:00,067
	Native	0:00,067
	Titanium	0:00,067
5	Xamarin	0:00,083
7	RN	0:00,117

iOS
network full render

approx. frames	tool	time
9	Flutter	0:00,150
11	Native	0:00,183
15	RN	0:00,250
23	Xamarin	0:00,383
39	Cordova	0:00,650
68	Titanium	0:01,150

Table 5: Loading a list of items both locally and from the network on Android and iOS, sorted by approximate number of frames. Timing starts at the input to open the new screen being given, and for the transition ends when the old screen disappears, and for the full render when the first screenful of list items with their images are visible.

Worth noting is also that React Native on Android was the only one of the twelve test apps to flash the screen to white between the screens when doing the transition. This one frame flash was noticeable to naked eye, so hopefully this was just a side-effect of pushing the animation configuration to its limit in the direction that most real apps don't do.

As loading the item list and the images might take a while, it can potentially create a situation where the new screen is empty before the list is loaded and ready to be shown.

This is better than having the app sit at the old screen before swapping to the fully loaded list screen, as it gives feedback to the user that the input was registered, and a loading indicator can be shown on the new screen during that time. This is why two separate times are shown for each test in Table 5, so we can compare the screen transition times to ones in Table 4 to see if the loading slows down the transitions and to see how long it takes to display the first screenful of content from the list.

The screen transition times took quite a hit on the local versions of the tests where JSON file and images were loaded from the device disk. On Android at least part of it was again from the screen recording using the disk at the same time, but other than that it seems a bit odd. There are two possible explanations, but neither of the them have been confirmed. The first is that threading is handled differently for the two sources of data, with networking being always moved to a background thread but some of disk I/O being done on whatever thread does the call. The way how Titanium behaved suggests that this could be the case with it, as on both platforms the screen transitions straight to the fully loaded list with local data source but not with the network one. The second possible explanation is that maybe the JSON and images were loaded from the disk on another thread, but the read from disk finished before the transition to an empty screen happened and the results were handled on the UI thread, slowing down the transition. Even though the screen transition was slower in the local data source tests, displaying the list with its images was still faster with the local data for all apps except the iOS React Native one.

The list loading test had probably the largest spread of results that can't be explained by timing inaccuracies, screen recording affecting some apps more than others or by some of the Android apps using 32-bit binaries. Flutter beat or tied native implementations on both platforms in all categories, but other cross-platform apps seemed a bit weak against the native apps, especially with showing the final list from network source. Cordova transitioned to an empty screen while loading the network data faster than it transitioned to a screen with a static layout but wasn't too impressive with finally filling that empty screen with the list. Xamarin was oddly inconsistent between platforms, with the Android version sticking out with its bad performance.

7.4. Reacting to a button press

Android

approx. frames	tool	time
0	Flutter	0:00,000
	Native	0:00,000
	Titanium	0:00,000
	Xamarin	0:00,000
1	Cordova	0:00,015
4	RN	0:00,065

iOS

approx. frames	tool	time
0	Native	0:00,000
1	Cordova	0:00,017
	Flutter	0:00,017
	RN	0:00,017
	Titanium	0:00,017
	Xamarin	0:00,017

Table 6. Fastest reactions to a button being pressed on Android and iOS, sorted by the approximated number of frames. Timing is from the input to start the task being given to a text being updated on screen.

Table 6 shows that most of the apps had no problems with input latency. The 0-frame reaction on iOS Native was the only such occurrence in these specific tests, but Titanium also achieved it during the computation tests. It's hard to say if the 0-frame reactions were glitches in the recordings, delayed responses on the Assistive Touch marker or just needed so many unknown conditions to be met that they seem rare and random. Cordova, Flutter and React Native had also 2-frame reactions on iOS, so there was a bit more variance in the data than what the fastest results show.

On Android the 0-frame reactions were more common, but only native and Titanium got the perfect results with others having 1- and 2-frame results in the mix. React Native was clearly the slowest here and the several frames of reaction time was also present in the video camera results, confirming that this was not caused by the screen recording software.

7.5. Heavy computation

Android

approx. frames	tool	time
52	Native	0:00,869
80	Titanium	0:01,355
94	Cordova	0:01,595
867	Xamarin	0:14,724
3411	RN	0:57,986
7868	Flutter	2:13,742

iOS

approx. frames	tool	time
53	Native	0:00,900
	Xamarin	0:00,900
76	Flutter	0:01,284
1616	Cordova	0:27,470
1618	Titanium	0:27,504
1637	RN	0:27,821

Table 7. Fastest completions of finding all primes below 3 000 000 on Android and iOS, sorted by the approximated number of frames. Timing is from the input to start the task being given to a text informing about the computation being finished appearing on the screen.

The computation test results shown on Table 7 turned out differently than originally expected on Android. Flutter and Xamarin were expected to be quite fast with their

compiled code but instead struggled with the test. React Native's JavaScript performance was also unexpectedly slow while the other two apps running JavaScript didn't have any problems. The slow apps were slow in the video camera tests too, though the screen recording did exaggerate the problem. For example, on Android Flutter's video camera times were between 1:55 and 2:14 and the screen recording times between 2:13 and 2:46.

The slowdown problems with Flutter, React Native and Xamarin most likely had to do with the fact that all three of them still used 32-bit binaries by default on Android when the final APKs were created. Running an app with 64-bit binaries would give it access to more memory, and more importantly it could handle more data per CPU cycle and be able take advantage of other features of the more modern 64-bit ARMv8 architecture. Flutter has 64-bit support, but the default release APK is built with only 32-binaries [marnberg, 2018], and as enabling the 64-bit build wasn't covered in the documentation [PAAAFR, 2019], the option wasn't used in the APK used in testing. Xamarin's support for 64-bit runtime was at the time "experimental" [CA, 2018] and not enabled by default, so it wasn't enabled for the testing. React Native didn't have 64-bit support at all when the APKs were finalized. The debug version of the Flutter app running its Dart code in interpreted mode ran the test in just couple of seconds, which made the results with the production APK even more interesting.

The iOS results with the 3 compiled languages being quite close with good results and the JavaScript languages getting almost identical result was expected, though the performance of JavaScriptCore, the iOS SDK built-in JavaScript interpreter, left lot to be desired compared to what Cordova and Titanium could do on Android.

The Titanium implementation didn't meet the specifications of the test as it was run on the main JavaScript thread and thus blocked the UI. This was due to the main Titanium SDK not having multithreading support and the external worker thread package received Android support only after the APKs were finalized [APWIFTTM, 2019]. React Native didn't have official multithreading support either, so a third-party package called react-native-threads was used (https://github.com/joltup/react-native-threads).

The Cordova app took a bit of a shortcut on iOS as the default progress bar on iOS webviews isn't animated like on Android, so it didn't have the animation mentioned in the specification. This wasn't noticed before analysis of the videos had started and didn't seem to be big enough issue to justify redoing the tests, as the animation was added to showcase that the UI isn't blocked during the computation and that could be confirmed without redoing the tests.

Some of the apps were slower than normal to update the screen when the computational task started. The iOS Cordova app took 8 frames at its fastest to show that the computation had started, and on Android Flutter's best was 14 frames and React Native's 7 frames. The slow responses were likely due to costs of creating the second runtimes that enable their single-threaded languages to do parallel execution.

7.6. Triggering vibration

Android		iOS	
time	tool	time	tool
0:00,041	Cordova	0:00,077	Titanium
0:00,048	Flutter	0:00,093	Cordova
0:00,062	Native	0:00,095	Native
0:00,116	Xamarin	0:00,096	Xamarin
0:00,147	Titanium	0:00,097	RN
0:00,159	RN	0:00,109	Flutter

Table 8. Fastest times to trigger vibration on Android and iOS, sorted by time. Timing is from the input to start the task being given to sound of vibration starting.

The vibration triggering test was given earlier as a prime example of a test that is flawed with the logging-based measurements, but it seems to be problematic with video-based measuring as well. The results in Table 8 don't really make sense with native implementations losing to cross-platform implementations that had go through multiple hoops to do the same function call as the native one. There might be a case for Cordova to be faster than native on Android as Chromium has native web vibration API support, but all of the other cross-platform apps should have been at a clear disadvantage. The spread of results was also much larger than expected, especially on Android. Most likely culprit for inconsistent results is the vibration motor being an inconsistent audio source, but the audio/video synchronization can also affect the results. Maybe with a larger dataset these results could be proven to be accurate, but with three samples per app these results are written off as invalid due to the timing method not being a good fit for this test.

7.7. Frozen frames during scroll

Android		iOS	
approx. frames	tool	approx. frames	tool
0	Cordova	**1**	Cordova
	Flutter		Native
	Native		RN
	Titanium		Titanium
4	RN	**2**	Xamarin
56	Xamarin	**3**	Flutter

Table 9. Least frozen frames during scrolling through list of items with images and short text on Android and iOS, sorted by the approximated number of frames.

The scrolling lag frames test was done on the network version of the list items screen. The range from which the frames were counted was from the first frame of movement to 20 frames before the moving ended. The last 20 frames were taken out as many apps had

multiple doubled frames in that range, but that was likely due to the movement slowing down to sub-pixel speed per frame rather than the app not being able to keep up.

The results on Table 9 show that most apps could do the scrolling just fine. Most of the frame drops were at the very beginning of the scroll and in normal use would've gone unnoticed as the user would have still focused on the swipe while the frame drops occurred. This however doesn't apply to Xamarin on Android which had serious slowdown issues, including many consecutive dropped frames (the same frame being shown for more than 2 times in a row) that made the scroll look even worse. This result becomes even more puzzling considering that on iOS Xamarin didn't have this problem and worked just as well as all the other tools. Xamarin's documentation and samples showcased some settings that could be tweaked on how the list item's layout was created or cached, but none of them seemed to do noticeably better than the one used in the final APK.

Getting these numbers from a variable frame rate video was hard, as both identical frames in the video and frames that were shown for over 33 milliseconds needed to be counted separately. The frame timing analysis might also be inaccurate without knowing how the rendering of these devices with vertical sync and other possible frame buffers works, as multiple frames that take between 18 to 32 milliseconds to render in a row must eventually cause a frame drop that might go unnoticed with this type of analysis.

8. Discussion

Both the results of the comparison study and the experiences on collecting the dataset by recording videos of a device running an app have lots of interesting and even some surprising aspects to them. This chapter is separated to three sections that analyze the results in general, impressions on the tools and video-based performance measuring separately to keep it structurally sound and easier to follow.

8.1. General results

To summarize the results, one can say that cross-platform apps do worse in performance benchmarks than native apps, but in most cases the performance hit isn't significant enough to truly harm the user experience. The results also contain some points that are worth discussing a bit more, such as the cases where the performance hit can be significant and differences between the two platforms.

The tests that made most of the cross-platform apps look bad compared to their native counterparts were the app launch and new screen opening tests on Android and computation test on iOS. Technically the computation test on Android was also bad for three of the apps, but the probable cause for those results was already stated (the 32-bit binaries) and will be discussed more below. Computation being slow on the cross-platform side of the app shouldn't be a problem in most cases, as that can in many cases be offloaded to a server or be ran on the native side. The launch times and new screen transitions being slow on the other hand don't have similar easy solutions to them and are the ones that can hurt the user experience the most, which makes them quite problematic. If the first interaction with the app (launching it) takes almost two seconds (compared to native app's half a second), the user session isn't off to a great start. Google's research data shows that visitors are more likely to abandon a website if it's slow to load [An, 2018], and if we draw a parallel between website loads and app launches & screen transitions, it shouldn't be too far reaching to suggest that the app being slow might eventually cause enough frustration that the user stops using the app or uninstalls it. Times for transitioning to a screen with a list of items were also somewhat bad for most of the cross-platform apps on both platforms, but the problem can be reduced by postponing the loading the list's content till the screen transition has begun and by showing a loading indicator. This of course doesn't remove the fact that loading and showing the list is slow, but makes the app seem more responsive. As the list loading times stayed generally under one second, the retention rate shouldn't be too affected as we're safely under the Google's recommendation of keeping site load times under three seconds [An, 2018]. Though we should also remember that the list layout used in the tests was very simple, so having a more complex layout might push the loading times to retention dropping territory.

The 32-bit binary problem that Flutter, React Native and Xamarin had on Android might need a short explanation of what it is and why it matters. Android gained support for 64-bit ARM processors in 2014 with the 5.0 release when support for the armeabi-v8a ABI (Application Binary Interface) was added [AL, 2019]. The 32-bit binaries are mostly built for the armeabi-v7a ABI, which has been supported since 2010 [NDKRH, 2019]. So, on top of the typical 64-bit benefits of having access to more memory and being able to handle more data per CPU cycle, the 64-bit binaries also have access to 4 years' worth of advancements in CPU instructions and features. 32-bit compilers might also just lack important optimizations, like in Flutter's case the 32-bit Dart compiler generated unoptimized code for MD5 hashing [MD5C, 2018]. Apple introduced support for 64-bit apps to iOS with iPhone 5S in 2013 [Clover, 2017], and with its tight control of the iOS ecosystem it transitioned completely to 64-bits quickly by first demanding new apps to have 64-bit support since beginning of 2015 and removing 32-bit support completely from iOS 11 onward (released in fall 2017) [Clover, 2017]. Google has been slower with the transition, with 64-bit support becoming mandatory for new apps and updates to older apps on Play Store in August 2019 [Cunningham, 2017]. With no external push to move to 64-bits and the 32-bit only Android 4.4 still being relatively widely used [DD, 2018], Facebook and Microsoft were quite slow to add the support as a non-experimental feature. React Native got its 64-bit support in the 0.59 release in March 2019 [Turner, 2019] and Xamarin's defaults were changed to include 64-bit binaries in March 2019 as well [Peppers, 2019]. Flutter has had 64-bit support for a longer time, but the default behavior is to build APKs with only 32-bit binaries, which is likely done to keep APK sizes smaller while supporting all devices. The newer Android App Bundle format also solves the APK size issue for apps published to the Play Store as it supports delivering customers different APKs with only the binaries relevant to their devices, making Google even less incentivized to change the default behavior of Flutter's APK packaging. But now that all three of the tools have their 64-bit support released, the performance issues found in these tests shouldn't be an issue anymore as long as newest versions of the tools are used and configured correctly.

Comparing the absolute times on Android and iOS directly wouldn't be fair due to the differences in how the screen recording software worked (storing to device's own disk vs. streaming the screen to a computer) possibly affecting the results, but we can still make some general observations. Two most obvious ones are that Android seemed to have higher highs and lower lows in the performance, and that on iOS all the apps had quite similar times in many tests. The native iOS app didn't seem to have too much of an edge over the cross-platform ones like its Android counterpart did. Between the three test runs for each test, iOS times were also less scattered than Android ones. Combining these observations together we can conclude that the cross-platform apps perform better on iOS and can more often reach native-like performance than on Android. We can't be sure for why this is, but the reasons are probably related to differences between the platforms in

general and the tools just being more optimized on iOS. We can't really know how the tools are optimized on each platform, and the only hint we have about Android possibly being neglected by the tool creators is the delayed 64-bit support for React Native and Xamarin.

8.2. Evaluating the tools

Let's go through all the impressions that each tool gave through the results and the benchmarking app implementation process. We'll go through the tools in alphabetical order, starting with Cordova. As the hybrid app tool representative in this comparison, it did quite well. Its biggest blunders were the iOS launch time, iOS computation speed and list loading performance on both platforms. Considering that many other tools also didn't do that well with lists either it can't be blamed too much for that. From these results we can say that using web technologies is, at least performance-wise, a valid option for creating mobile apps, but you'd still want to evaluate your options for which specific tool to use. For some creating a web app for use through a regular browser might suit better than creating an installable hybrid app, and some might want to use more complete app development tools like Ionic to get a head start on stylizing the app to look more native-like on each platform.

If we need to declare one of the tools as a winner of this comparison, the honor would go to Flutter. If we don't count the back navigation and Android computation tests, the only tests where Flutter significantly lost to native apps were the launch time tests on both platforms. It even beat the native apps in list loading performance, the test that most cross-platform apps struggled with. The biggest reasons why someone might want to steer away from trying or using Flutter come from reasons outside of performance, like not being familiar with the Dart language or not liking its APIs. One big flaw of those APIs is that there isn't any built-in abstraction for creating an app with a native look and feel on both platforms, but rather one would need to code the UI portion twice and try to share the business logic components to achieve that.

Titanium's overall performance in the comparison is hard to judge, as its results were quite mixed with some results competing or even beating native app's and some just being horrible. The results weren't even consistent between platforms, as it beat the native iOS app in launch time but on Android it took over two seconds. Overall it ended up being in the middle of the road and other than the Android launch times and the iOS list loading times there aren't any showstoppers for using it from performance perspective. Titanium was the only tool in the comparison that needed platform specific code to implement the benchmarking app, as setting up navigation between screens required a NavigationWindow widget on iOS, but using it wasn't supported on Android. This meant that parts of the UI defining XML needed to be duplicated and the navigation logic also needed some platform specific adjustments. With such an important feature that almost

all apps use not having a cross-platform implementation, overall lack of up-to-date community resources when comparing to other tools and other minor headaches that came up while developing the app, Titanium doesn't really have anything going for it and it's hard to recommend over other tools.

Despite being one of the more popular cross-platform tools during last couple of years, React Native's performance was a bit disappointing. The only test where it stood out in a positive light was the iOS launch time test, being second and beating the native app by 5 frames. Other than that, React Native was consistently in the slower half of the group and in many tests got the straight up slowest times. Some of the results might have come from problems with the third party React Navigation library's implementation, but as long as that library is recommended by the official documentation [NBS, 2018], that's how lot of people will experience React Native's screen transitioning performance. On iOS it placed slightly better than on Android, so we'll see how the 64-bit support and upcoming big threading model changes will affect its performance. With these results it's obvious that React Native can't be recommended for developers looking for the best performance and snappy response times, especially when counting in the lack of official multithreading support. However, for web developers already familiar with ReactJS on the web it can be very appealing to jump to mobile app development with React Native, as they are already familiar with its core concepts and can jump right in to development.

Xamarin had lot of unexpected small performance issues. As a representative of the cross-compiled tools group, it was expected to perform with almost native level performance, but it couldn't meet those expectations and lost to native and some other cross-platform tools in most of the tests. While some of the issues could have potentially come from unoptimized code, they can't be completely discarded as false alarms as most of the code was done by following samples in the documentation. This means that either the tool itself doesn't perform too well or official samples make it perform bad, and neither of these situations are good for Xamarin. With its slow launch times and bad list performance on Android, it can't really be recommended even for the simple CRUD apps which should be the best use case for cross-platform tools with little to no need for accessing device features. Similar to ReactJS developers possibly finding React Native interesting, the main group that could still find Xamarin attractive are developers that have previously worked with .NET products and other Microsoft's tools. Those people would have a head start in the learning process and could potentially find more optimal implementations and thus fixing the performance issues that came up in this study.

8.3. Video-based performance measuring

As many previous studies on cross-platform app performance had used the timestamped logs approach of measuring time [Corral *et al.*, 2012; Willocx *et al.*, 2015], the newly adopted the frame-by-frame video analysis method inspired by Soomro's

[2015] video game input latency testing used in this study is just as much worth of discussion as the results themselves. The two methods have their own pros and cons, and the better method can change on a test by test basis. The video analysis method can be divided to two categories depending on the video capture method used: high-speed video camera or by recording the device screen. Selecting which one to use needs some serious consideration as they have quite significant tradeoffs.

The biggest flaw of the logging-based timing method is that we can't be sure if the times between tools are comparable when the test depends on the UI or the bridging mechanism is involved. As previously explained, each tool has its own overhead on delivering the native UI events to app specific code for us to do the logging nor can we reliably use logging when timing should start or end on the native side of the bridge where developers don't have access. For tests that don't fall into those two groups, logging can be superior to video analysis. For example, the computation test in this study could have been timed with logs with no issue, and in fact would've benefited from the better accuracy as the two best times on iOS tied with the video method. Though still even in these cases doing some observation of what is happening on the screen can be beneficial, evidenced by the slow button response of some apps in the computation tests. Processing the individual logs to a proper dataset can also easily be automated, which isn't the case for video analysis, allowing collection of larger dataset in a reasonable amount of time.

The video-based timing addresses the flaws of logging by not relying on the code at all for the timing. It doesn't matter if we don't have access to the best place to insert our timing code because we can accurately time anything that can be picked up from a video. It also opens up new types of tests that are completely based on the UI, such as counting the amount of dropped frames during scrolling animation. The tests could also be run on apps not specifically made for testing, so someone could check how "real" publicly distributed apps perform against these small purpose made benchmarking apps and see if the results on benchmarking apps apply to production apps.

In this study, the biggest downsides of timing with videos were difficulty of transforming the raw videos to datasets that could be analyzed and technical difficulties in capturing the footage. With properly designed and formatted logs, one could easily write a script that takes the raw device logs, finds appropriate lines from them and writes the results into a csv file or something similar that is easy to process further. With videos, automatically determining the timing start- and endpoints isn't as straightforward as you have the frame images and the time when they were shown to consider. With proper planning however, it should be possible to automate the process for many types of tests, especially ones without animations. If the apps were to change large portions of the screen, like the background color of the whole screen, when timing should start and end, a single script could easily find the proper frames and then look up their timestamps. The benchmarking app wasn't designed to be friendly for such scripts and made only small changes to the UI, which were all different for each test and for each app happened in

slightly different positions on the screen. This means that a script would need to be adjusted for each test to look at specific pixels, which in the worst case could be different for starting and ending the timing, adding back lot of the manual labor of going through the videos by hand.

Out of the two methods of capturing the video footage, the screen recording software method was easier to use after initial setup, and the video were of better quality. The problems with screen recording are in the implications it has on the performance and in the variable framerate videos that the software used in this study produced. The effects on performance were also different between platforms due to the differences on how the screen recording software worked (storing locally vs. streaming to a computer). The flaws in the high-speed camera footage however made using the screen recording software the better choice in this study. As newer devices have the capability of connecting to a HDMI monitor with adapters, if that video signal is of high quality and 60-fps on both platforms then problems with both variable framerate footage and platform specific performance hits could be circumvent by connecting the phones to a capture device with HDMI. If this were the case, then the screen recording method could possibly be the preferred method over high-speed camera method, not just the method that needs to be used because the alternative has crippling flaws. This would be due to smaller costs (a couple of adapters versus a good camera & other equipment), no need for special recording space (camera needs proper lighting and a stand) and automating the analysis of video material being easier on the screen recording material that doesn't have as much noise or is affected by the screen used in the device running the app.

Making the high-speed camera recordings a viable option would require a lot of equipment, design and testing. A professional grade video camera would be required to get better quality footage and to have more options on the framerate. The Nexus 6P used as the camera in this study also marked the recordings as variable framerate footage, which would be another reason to switch to a professional camera to make sure all the frames in the recording are equal in length and make the analysis easier. Even if the image quality problems can be solved with better off-the-shelf equipment, determining the exact moment of input being given doesn't have any easy solutions. The trick used in this study was to place a custom-made stylus that didn't cause an input event by itself on the screen, and then touch it with a piece of metal that would cause the screen to register an input. This made determining the frame of input being given easier than using a regular stylus because the direction of the movement in relation to the camera but there were still many situations where it was hard to pick a specific frame as the one where the touch happened. It also required two hands, obstructing lot of the screen from the camera. An ideal solution would be a stylus that powers up a LED light when it touches the screen, but there didn't seem to be any available on the market when testing was planned. If one were to make one by themselves, they could make the light to be triggered either by the pressure of touching the screen or by the electricity in the screen. Automating this type of video

capturing process would require at least a robotic hand on top of all the other equipment, which starts to make this method to be too unpractical to setup for one-off studies.

Using audio as part timing in the video tests doesn't seem to be a valid option. In the test results this can be seen in the vibration tests results, where the spread of times was unexpectedly large both on per platform and per app basis, and the native apps being outsped on both platform even though they should've had a clear advantage with their direct access to the vibration APIs. During the initial planning and testing of how to do the high-speed camera recordings, one idea to overcome the issue of seeing accurately when the touch happened was to use the sound of a stylus hitting the touchscreen instead of trying to use the video. It didn't take too many test runs to find a video where the timing ending screen update happened before the camera's microphone picked up the sound, and there is also the issue of the distance between the camera and the stylus possibly affecting the results. What we can conclude from this is that mixing different signals, like video and audio, to do the timing should be avoided at all cost. This means that in future studies the vibration test would need to be replaced by some other test that can be used to showcase the tools' speed of accessing platform specific features completely through the video. Some possible replacement tests could be showing a notification, opening another app (like when opening a link in the default browser app) or turning on the device camera's flash on.

9. Conclusion

Is using a cross-platform tool to create your app a surefire way to kill its performance and make it "janky"? For regular use cases that aren't that demanding in the first place, no. According to these results using them is fine, especially if you have the knowledge on how to properly use the tool and optimize it properly (Xamarin's list performance on Android). Though for some the fact that you can't just put something together and have it work flawlessly on both platforms can be frustrating. For example, Airbnb talks of their venture into using React Native as needing to support three platforms instead of two [Peal, 2018b] and have since started moving back to fully native approach on both Android and iOS.

If we were to pick a winner from the comparison, it would be Flutter, as it was the only tool that didn't struggle with any of the tests, if we're not counting the Android computation test due to the fix being passing a single flag to the compiler. It's due to that side note and other similar ones in the testing setup and benchmarking app definitions not allowing the apps to perform at their absolute best that these result can't be called conclusive or absolute. What would've the results have been if all the tools could've used all the available, even experimental, optimizations? What would the results have been if the default animations and button behavior were used instead? Some future study could revisit these tests now that the Android 64-bit support has been introduced to all of the tools and try to define the tests in a way that allows all the apps to work with their default behavior.

Some ways of how performance critical apps might use cross-platform concepts in the near future are shared architecture & design and Kotlin multiplatform. The server-driven UI that Airbnb currently uses [Peal, 2018a] and the RIBs architecture developed at Uber [Tran and Zhu, 2016] are some examples of creating models that allow developers to share resources and design patterns across platforms while still implementing the apps with native SDKs. Kotlin multiplatform is JetBrains' initiative to make sharing code between all platforms that support Kotlin easier, which include Android and iOS [MP, 2018]. With Kotlin's great interoperability with other languages, one could use a single Kotlin codebase to write all but the UI layer of the app in pure Kotlin, and then create the UI for each platform separately mixing Kotlin and platform's default language(s). Apps created with Kotlin multiplatform would fall in the traditional cross-compiled app category in Raj and Tolety's [2012] list of cross-platform app patterns, but if someone were to create a cross-platform UI library on top of it, it would create a new branch of cross-compiled category like Flutter's embedded rendering engine approach did.

The video-based method of measuring app performance proved to be effective, but not without its problems. It allows measuring app's performance as users perceives it at the cost of adding manual steps to collecting the dataset when compared to the old timestamped logs method. The two methods of capturing the video material, using a high-speed video camera or screen recording software, both have their problems that

complicate doing the measuring this way. Future studies should take their time to improve the chosen video capturing process to address these issues, with some possible things to try being using a better video camera and testing the HDMI outputs of newer smartphone models. As part of improving the process automation of analyzing the videos should also be tested to help scale up the size of datasets that can be collected. While this type of situation where apps under testing are visually similar but have different internals is probably the prime example for where video-based testing can be used, there are other types of situations too where it could be tried out, like testing apps downloaded from the App Store or the Play Store. It doesn't even need to be limited to smartphones or even computers, as with the camera method any UI with visual responses could potentially be tested this way.

References

[AAATAN, 2011] Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap. Press release, October 3rd, 2011. https://news.adobe.com/press-release/adobe-creative-cloud-dps/adobe-announces-agreement-acquire-nitobi-creator-phonegap (Accessed October 28th, 2018)

[Ahti *et al.*, 2016] Ville Ahti, Sami Hyrynsalmi and Olli Nevalainen. An Evaluation Framework for Cross-Platform Mobile App Development Tools: A case analysis of Adobe PhoneGap framework. In: *Proc. of the 17th International Conference on Computer Systems and Technologies 2016*, 41-48.

[Allen, 2011] Jonathan Allen. The death and rebirth of Mono. News article, May 17th, 2011. https://www.infoq.com/news/2011/05/Mono-II (Accessed October 20th, 2018)

[Alpert, 2018] Sophie Alpert. State of React Native 2018. Blog post, June 14th, 2018. https://facebook.github.io/react-native/blog/2018/06/14/state-of-react-native-2018 (Accessed October 28th, 2018)

[Al-Heeti, 2018] Abrar Al-Heeti. Oracle v. Google ain't over yet -- Google vows it'll appeal to Supreme Court. News article, August 28th, 2018. https://www.cnet.com/news/oracle-v-google-aint-over-yet-google-vows-itll-appeal-to-supreme-court/ (Accessed October 28th, 2018)

[An, 2018] Daniel An. Find out how you stack up to new industry benchmarks for mobile page speed. Online article, updated Feburary 2018. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/ (Accessed March 18th, 2019)

[Andrade and Albuquerque, 2015] Paulo R. M. de Andrade and Adriano B. Albuquerque. Cross Platform App: A Comparative Study. *International Journal of Computer Science & Information Technology (IJCSIT)*, 7 (1), 33-40. doi: 10.5121/ijcsit.2015.7104

[AL, 2019] Android Lollipop. Developer documentation. https://developer.android.com/about/versions/lollipop (Accessed March 18th, 2019)

[APWIFTTM, 2019] Android: Add parity with iOS for the Ti.Worker module. Software development tracker, updated January 2019. https://jira.appcelerator.org/browse/MOD-2351 (Accessed February 9th, 2019)

[Angulo and Ferre, 2014] Esteban Angulo and Xavier Ferre. A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX. In: *Proc. of the XV International Conference on Human Computer Interaction*, Article 27, 8 pages.

[Barney, 2008] Lee Barney. QuickConnect iPhone: an iPhone UIWebView hybrid framework. Blog post, May 28th, 2008. https://tetontech.wordpress.com/2008/05/28/quickconnect-iphone-an-iphone-hybrid-framework/ (Accessed August 15th, 2018)

[Bort, 2013] Julie Bort. Microsoft Might Buy A Startup That Powers 10 Percent Of The World's Smartphones. News article, February 1st, 2013. http://www.businessinsider.com/microsoft-eyes-appcelerator-acquisition-2013-2#ixzz2YmNSFhT7 (Accessed August 15th, 2018)

[RNChangelog, 2019] Changelog. Software development tracker, updated February 28th, 2019. https://github.com/react-native-community/react-native-releases/blob/master/CHANGELOG.md (Accessed March 8th, 2019)

[Chedeau, 2015] Christopher Chedeau. From Hackathon to React Native. Online video, September 20th, 2015. https://www.infoq.com/presentations/react-native-facebook (Accessed February 9th, 2019)

[Ciman and Gaggi, 2017] Matteo Ciman and Ombretta Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. Pervasive and Mobile Computing, 39, 214-230

[Clover, 2017] Juli Clover. 32-Bit Apps No Longer Supported in iOS 11. News article, June 6th, 2017. https://www.macrumors.com/2017/06/06/32-bit-apps-no-longer-supported-in-ios-11/ (Accessed March 17th, 2019)

[Corral *et al.*, 2012] Luis Corral, Alberto Sillitti and Giancarlo Succi. Mobile multiplatform development: An experiment for performance analysis. Procedia Computer Science, 10, 736-743.

[CA, 2018] CPU Architectures. Developer documentation, March 1st, 2018. https://docs.microsoft.com/en-us/xamarin/android/app-fundamentals/cpu-architectures (Accessed February 9th, 2019)

[Cunningham, 2017] Edward Cunningham. Improving app security and performance on Google Play for years to come. Blog post, December 19th, 2017. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html (Accessed February 8th, 2019)

[DD, 2018] Distribution Dashboard. Developer documentation, data from October 26th, 2018. https://developer.android.com/about/dashboards (Accessed February 8th, 2019)

[Empiricalis, 2018] empiricalis. Re: Facebook moving away from React Native? Online discussion comment, June 3rd, 2018. https://www.reddit.com/r/androiddev/comments/8o4p6n/facebook_moving_away_from_react_native/e00vy4x/?st=jtdfpfjl&sh=7d8e007f (Accessed October 22nd, 2018)

[FJRRN, 2018] Facebook just release react-native 0.56 which is totally broken on windows. Online discussion thread, 2018. https://www.reddit.com/r/reactnative/comments/8w8qsd/facebook_just_released_reactnative_056_which_is/?st=jnlxl4cy&sh=b0d4164d (Accessed October 23rd, 2018)

[Fei-Fei *et al.*, 2004] L. Fei-Fei, R. Fergus and P. Perona. Workshop on Generative-Model Based Vision. CVPR 2004.

[Ferreira *et al.*, 2018] Cristiane M. S. Ferreira, Maria J. P. Peixoto, Paulo A. S. Duarte, Andrei B. B. Torres, Messias L. Silva Júnior, Lincoln S. Rocha and Windson Viana. An Evaluation of Cross-Platform Frameworks for Multimedia Mobile Applications Development. IEEE Latin America Transactions, 16 (4), 1206-1212

[FSA, 2017] Flutter System Architecture. Developer documentation, April 23rd, 2017. https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/edit#slide=id.gbb3c3233b_0_187 (Accessed August 25th, 2018)

[Gregavola, 2012] gregavola. Re: Scrolling Performance in WebView for Android - Jelly Bean 4.1.x. Online discussion comment, July 26th, 2012. https://stackoverflow.com/a/11669972 (Accessed October 23rd, 2018)

[Guthrie, 2016] Scott Guthrie. Microsoft to acquire Xamarin and empower more developers to build apps on any device. News article, February 24th, 2016. https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/ (Accessed October 24th, 2018)

[Haynie, 2016] Jeff Haynie. Axway Acquires Appcelerator — And Why This is Great News for All. Blog post, January 17th, 2016. https://www.appcelerator.com/blog/2016/01/axway-acquires-appcelerator-and-why-this-is-great-news-for-all/ (Accessed October 20th, 2018)

[Heitkötter *et al.*, 2013] Henning Heitkötter, Sebastian Hanschke and Tim A. Majchrzak. Evaluating Cross-Platform Development Approaches for Mobile Applications. In: *International Conference on Web Information Systems and Technologies*, 120-138.

[Humayoun *et al.*, 2013] Shah Rukh Humayoun, Stefan Ehrhart and Achim Ebert. Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study. In: *International Conference on Human-Computer Interaction*, 371-380.

[Hyperloop, 2019] Hyperloop. Webpage. https://www.appcelerator.com/mobile-app-development-products/hyperloop/ (Accessed February 8th, 2019)

[Illbuyajuicer, 2018] illbuyajuicer. Re: Flutter or Kotlin, Which one is the future? Online discussion comment, June 1st, 2018. https://www.reddit.com/r/androiddev/comments/8ntbky/flutter_or_kotlin_which_one_is_the_future/dzyp5eg/ (Accessed October 22nd, 2018)

[IXSTB, 2018] Is Xamarin still that bad? Online discussion thread, 2018. https://www.reddit.com/r/androiddev/comments/8dfdzx/is_xamarin_still_that_bad/ (Accessed October 22nd, 2018)

[Johnson, 2008] Dave Johnson. PhoneGap: It's Like AIR for the IPhone. Blog post, September 18th, 2008. https://phonegap.com/blog/2008/09/18/phonegap-its-like-air-for-the-iphone/ (Accessed October 28th, 2018)

[JSXID, 2019] JSX In Depth. Developer documentation, updated February 7th, 2019. https://reactjs.org/docs/jsx-in-depth.html (Accessed February 8th, 2019)

[Kevin, 2016] Kevin. Re: Cordova android scrolling/performance issues. Online discussion comment, April 27th, 2016. https://stackoverflow.com/a/36880002 (Accessed October 23rd, 2018)

[Krill, 2009] Paul Krill. Appcelerator enables iPhone, Android app dev. News article, June 8th, 2009. https://www.infoworld.com/article/2632710/application-development/appcelerator-enables-iphone--android-app-dev.html (Accessed August 15th, 2018)

[Knöchel, 2016] Hans Knöchel. Re: Appcelerator Hyperloop vs. Plain Titanium Modules. Online discussion comment, August 16th, 2016. https://stackoverflow.com/questions/38951699/appcelerator-hyperloop-vs-plain-titanium-modules/38970656#38970656 (Accessed October 20th, 2018)

[LouisCAD, 2016] LouisCAD. Instance state not saved when app is killed by OS. Online discussion opening comment, November 12th, 2016. https://github.com/flutter/flutter/issues/6827 (Accessed October 22nd, 2018)

[marnberg, 2018] marnberg. Build APK for multiple target platforms. Online discussion opening comment, July 11th, 2018. https://github.com/flutter/flutter/issues/19275 (Accessed February 9th, 2019)

[MD5C, 2018] MD5 compute hash file stronger slow in release mode. Online discussion thread, started December 1st, 2018. https://github.com/flutter/flutter/issues/24906 (Accessed February 9th, 2019)

[Mercado *et al.*, 2016] Iván Tactuk Mercado, Nuthan Munaiah and Andrew Meneely. The Impact of Cross-Platform Development Approaches for Mobile Applications from the User's Perspective. In: *Proc. of the International Workshop on App Market Analytics*, 43-49.

[MOSMSA, 2018] Mobile Operating System Market Share Asia. Webpage, data from 2018. http://gs.statcounter.com/os-market-share/mobile/asia/2018 (Accessed August 14th, 2018)

[Montemagno, 2017] James Montemagno. Share UI Code in any iOS and Android App with .NET Embedding. Blog post, December 11th, 2017.

https://devblogs.microsoft.com/visualstudio/share-ui-code-in-any-ios-and-android-app-with-net-embedding/ (Accessed March 28th, 2019)

[Mudiyala, 2017] Mudiyala. Xamarin.iOS file size is too big. Tried all the possible options. Online discussion opening comment, August 18th, 2017. https://forums.xamarin.com/discussion/103470/xamarin-ios-file-size-is-too-big-tried-all-the-possible-options (Accessed October 23rd, 2018)

[MP, 2018] Multiplatform Programming. Developer documentation, updated October 23rd, 2018. https://kotlinlang.org/docs/reference/multiplatform.html (Accessed March 7th, 2019)

[NM, 2018] Native Modules. Developer documentation, updated August 8th, 2018. https://facebook.github.io/react-native/docs/native-modules-android (Accessed October 25th, 2018)

[NBS, 2018] Navigating Between Screens. Developer documentation, updated December 2nd, 2018. https://facebook.github.io/react-native/docs/navigation (Accessed December 12th, 2018)

[Nelson, 2018] Randy Nelson. U.S. iPhone Users Spent An Average of $58 on Apps in 2017, 23% More Than the Year Before. Blog post, April 13th, 2018. https://sensortower.com/blog/revenue-per-iphone-2017 (Accessed August 14th, 2018)

[NS, 2018] Next Steps. Developer documentation, updated August 18th, 2018. https://cordova.apache.org/docs/en/latest/guide/next/index.html (Accessed February 7th, 2019)

[NDKRH, 2019] NDK Revision History. Developer documentation, updated 2019. https://developer.android.com/ndk/downloads/revision_history (Accessed March 8th, 2019)

[Ohrt and Turau, 2012] Julian Ohrt and Volker Turau. Cross-Platform Development Tools for Smartphone Applications. Computer, 45 (9), 72-79.

[OOTP, 2018] Out-of-Tree Platforms. Developer documentation, updated September 12th, 2018. https://facebook.github.io/react-native/docs/out-of-tree-platforms (Accessed October 28th, 2018)

[UTXMP, 2017] Part 1 - Understanding the Xamarin Mobile Platform. Developer documentation, updated March 27th, 2017. https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform (Accessed August 25th, 2018)

[Peal, 2018a] Gabriel Peal. What's Next for Mobile at Airbnb. Blog post, June 19th, 2018. https://medium.com/airbnb-engineering/whats-next-for-mobile-at-airbnb-5e71618576ab (Accessed February 8th, 2019)

[Peal, 2018b] Gabriel Peal. Sunsetting React Native. Blog post, June 19th, 2018. https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a (Accessed February 8th, 2019)

[Peppers, 2019] Jonathan Peppers. 13.3.2019. [Xamarin.Android.Build.Tasks] include arm64-v8a by default. Software development tracker, 2019. https://github.com/xamarin/xamarin-android/pull/2825 (Accessed March 20th, 2019)

[PS, 2018] Platform Support. Developer documentation, updated July 18th, 2018. https://cordova.apache.org/docs/en/latest/guide/support/index.html (Accessed March 19th, 2019)

[PAAAFR, 2019] Preparing an Android App for Release. Developer documentation, updated February 8th, 2019. https://flutter.io/docs/deployment/android (Accessed February 9th, 2019)

[Raj and Tolety, 2012] Rahul Raj C.P 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 (INDICON)*, 625-629.

[RE, 2019] Rendering Elements. Developer documentation, updated July 19th, 2018. https://reactjs.org/docs/rendering-elements.html (Accessed August 27th, 2018)

[RJSRouting, 2018] Routing. Developer documentation, updated October 26th, 2018. https://reactjs.org/community/routing.html (Accessed December 12th, 2018)

[IOSchedule, 2018] Schedule. Webpage, 2018. https://events.google.com/io2018/schedule (Accessed October 28th, 2018)

[Seidel, 2015] Eric Seidel. Sky: An Experiment Writing Dart for Mobile. Dart Developer Summit 2015. Online video, April 29th, 2015. https://www.youtube.com/watch?v=PnIWl33YMwA (Accessed October 28th, 2018)

[SE_API, 2019] Stack Exchange API. Web API. https://api.stackexchange.com/ (Accessed April 17th, 2019)

[Soomro, 2015] Adeel Soomro. Console Latency: Exploring Video Game Input Lag. Online article, 2015. https://displaylag.com/console-latency-exploring-video-game-input-lag/ (Accessed October 21st, 2018)

[FShowcase, 2019] Showcase. Webpage. https://flutter.io/showcase/ (Accessed April 17th, 2019)

[SIAIV, 2018] Support inlining Android/iOS views. Online discussion thread. https://github.com/flutter/flutter/issues/19030 (Accessed February 7th, 2019)

[SAXP, 2011] SUSE and Xamarin Partner to Accelerate Innovation and Support Mono Customers and Community. News article, July 18th, 2011. https://www.suse.com/c/news/suse-and-xamarin-partner-to-accelerate-innovation-and-support-mono-customers-and-community/ (Accessed October 20th, 2018)

[Taneja *et al.*, 2016] Kavita Taneja, Harmunish Taneja and Rohit K. Bhullar. Cross-platform application development for smartphones: Approaches and implications. In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, 1752-1758.

[Tran and Zhu, 2016] Vivian Tran and Yixin Zhu. Engineering the Architecture Behind Uber's New Rider App. Blog post, December 20th, 2016. https://eng.uber.com/new-rider-app/ (Accessed March 7th, 2019)

[Turner, 2019] Ryan Turner. Releasing React Native 0.59. Blog post, March 12th, 2019. https://facebook.github.io/react-native/blog/2019/03/12/releasing-react-native-059 (Accessed March 8th, 2019)

[UTMITL, 2018] Update to MIT license. Software development tracker, February 17th, 2018. https://github.com/facebook/react-native/commit/26684cf3adf4094eb6c405d345a75bf8c7c0bf88 (Accessed August 26th, 2018)

[WIARFNS, 2018] What is Android Runtime for NativeScript? Developer documentation, updated June 15th, 2018. https://docs.nativescript.org/angular/core-concepts/android-runtime/overview (Accessed October 19th, 2018)

[WURN, 2018] Who's using React Native? Webpage, updated March 6th, 2018. https://facebook.github.io/react-native/showcase.html (Accessed October 15th, 2018)

[WDTSHF, 2018] Why does this subreddit hate Flutter? Online discussion thread, started May 2nd, 2018. https://www.reddit.com/r/androiddev/comments/8gikul/why_does_this_subreddit_hate_flutter (Accessed October 22nd, 2018)

[Willocx *et al.*, 2015] Michiel Willocx, Jan Vossaert and Vincent Naessens. A Quantitative Assessment of Performance in Mobile App Development Tools. In: *2015 IEEE International Conference on Mobile Services*, 454-461.

[Xanthopoulos and Xinogalos, 2013]. Spyros Xanthopoulos and Stelios Xinogalos. A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications. In: *Proc. of the 6th Balkan Conference in Informatics*, 213-220.