A Study of Cordova and Its Data Storage Strategies

Gilles Callebaut*, Michiel Willocx†, Jan Vossaert†, Vincent Naessens†, Lieven De Strycker*

*KU Leuven
DRAMCO, Department of Electrical Engineering (ESAT),
Ghent Technology Campus, 9000 Ghent, Belgium
{gilles.callebaut, lieven.destrycker}@kuleuven.be
†KU Leuven
MSEC, imec-DistriNet
Ghent Technology Campus, 9000 Ghent, Belgium
{michiel.willocx, jan.vossaert, vincent.naessens}@cs.kuleuven.be

***Abstract*—The mobile world is fragmented by a variety of mobile platforms, e.g., Android, iOS and Windows Phone. While native applications can fully exploit the features of a particular mobile platform, limited or no code can be shared between the different implementations. Cross-Platform Tools (CPTs) allow developers to target multiple platforms using a single codebase. These tools provide general interfaces on top of the native Application Programming Interfaces (APIs). Apart from the performance impact, this additional layer may also result in the suboptimal use of native APIs. More specifically, this paper focuses on Apache Cordova; the most used CPT. Via a data storage case study, the impact of the abstraction layer is analyzed. Both the performance overhead and API coverage are discussed. Based on the analysis, an extension to the cross-platform storage API is proposed and implemented. In addition, the Cordova framework, including the employed bridge techniques, is studied and elaborated.**

Keywords–Cross-Platform Tools; data storage; performance analysis; API coverage; Apache Cordova/Phonegap.

I. INTRODUCTION

An increasing number of service providers are making their services available via the smartphone. Mobile applications are used to attract new users and support existing users more efficiently. Service providers want to reach as many users as possible with their mobile services. However, making services available on all mobile platforms is very costly due to the fragmentation of the mobile market. Developing native applications for each platform drastically increases the development costs. While native applications can fully exploit the features of a particular mobile platform, limited or no code can be shared between the different implementations. Each platform requires dedicated tools and different programming languages (e.g., Objective-C, C# and Java). Also, maintenance (e.g., updates or bug fixes) can be very costly. Hence, application developers are confronted with huge challenges. A promising alternative are mobile Cross-Platform Tools. A significant part of the code base is shared between the implementations for the different platforms. Further, many Cross-Platform Tools such as Cordova use client-side Web programming languages to implement the application logic, supporting programmers with a Web background.

Although several Cross-Platform Tools became more mature during the last few years, some skepticism towards CPTs remains. For many developers, the limited access to native device features (i.e., sensors and other platform APIs) remains an obstacle. In many cases, the developer is forced to use a limited set of the native APIs, or to use a work-around –which often involves native code– to achieve the desired functionality. This paper specifically tackles the use case of data storage APIs in Cordova. This paper extends our previous work [1] with an elaboration of the inner workings of the Cordova framework and new experiments concerning the employed bridge mechanisms.

Cordova is one of the most used CPTs [2][3]. It is a Web-to-native wrapper, allowing the developer to bundle Web apps into standalone applications.

Contribution. The contribution of this paper is fourfold. First, the Cordova framework including the Cordova Bridge is discussed. Second, four types of data storage strategies are distinguished in the setting of mobile applications (i.e., variables, databases, files and sensitive data). The support for each strategy using both native and Cordova development is analysed and compared. Third, based on this analysis a new Cordova plugin that extends the Cordova Storage API coverage is designed and developed. This plugin tackles a shortcoming in the currently available Cordova APIs. Finally, the security and performance of the different native and Cordova storage mechanisms is evaluated for both the Android and iOS platform.

The remainder of this paper is structured as follows. Section II points to related work. Section III gives an overview of CPTs. Section IV discusses the inner workings of Cordova applications, followed by Section V where an overview of data storage strategies and their API coverage in Cordova and native applications is given. The design and implementation of NativeStorage, a new Cordova storage plugin, is presented in Section VI. Section VII presents a security and performance evaluation of the Cordova and native storage mechanisms with a strong focus on the performance evaluation. The final section presents the conclusions and points to future work.

II. RELATED WORK

Many studies compare CPTs based on a quantitative assessment. For instance, Rösler et al. [4] and Dalmasso et al. [5] evaluate the behavioral performance of cross-platform applications using parameters such as start-up time and memory consumption. Willocx et al. [6] extend this research and include more CPTs and criteria (e.g., CPU usage and battery usage) in the comparison. Further, Ciman and Gaggi [7] focus

specifically on the energy consumption related to accessing sensors in cross-platform mobile applications. These studies are conducted using an implementation of the same application in a set of cross-platform tools and with the native development tools. This methodology provides useful insights in the overall performance overhead of using CPTs. Other research focuses on evaluating the performance of specific functional components. For instance, Zhuang et al. [8] evaluate the performance of the Cordova SQlite plugin for data storage. The work presented in this paper generalizes this work by providing an overview and performance analysis of the different data storage mechanisms available in Cordova, and comparing the performance with native components.

Several other studies focus on the evaluation of cross-platform tools based on qualitative criteria. For instance, Heitkötter et al. [9] use criteria such as development environment, maintainability, speed/cost of development and user-perceived application performance. The user-perceived performance is analyzed further in [10], based on user ratings and comments on cross-platform apps in the Google Play Store. The API coverage (e.g., geolocation and storage) of cross-platform tools is discussed in [11]. It is complementary with the work presented in this paper, which specifically focuses on the API coverage, performance and security related to data storage.

III. Cross-Platform Tools

Cross-platform tools can be classified in five categories based on their employed strategy [12]: JavaScript frameworks, Web-to-native wrappers, runtimes, source code translators and app factories. The latter provides a drag-and-drop interface to allow the creation of simple applications without programming knowledge. This paper focuses on the web-to-native wrapper Cordova. However, an overview of relevant CPT strategies is given to illustrate the differences, similarities and rationales. We can further classify cross-platform tool strategies in: (I) web-based CPTs and (II) source-code translators and runtimes.

A. Web-Based Cross-Platform Tool Strategies

JavaScript frameworks as well as Web-to-native wrappers make use of Web technologies for the development of mobile applications. A major advantage of these tools is that they enable Web developers to participate in mobile application development. Due to the availability of web browser capabilities in mobile operating systems this strategy is widely adopted. The user interface of such an application is developed with HTML and CSS, and the functionality is implemented using JavaScript.

To adapt to the specific interfaces and navigation patterns of mobile applications new mobile Javascript frameworks have been developed. These mobile interfaces are optimized for smaller screen sizes compared to regular websites and, for instance, provide support for the touch UI of mobile devices. Some frameworks also try to mimic the UI of native applications by providing native skins. Thereby tailoring the UI of the application to the look and feel of the platform on which it is running. These skins, however, do not provide a fully native experience. Most JavaScript frameworks also support traditional architectural design patterns such as Modelviewcontroller (MVC) and Modelviewviewmodel (MVVM) to facilitate the development of well-structured and maintainable code. Examples of such mobile JavaScript frameworks are: JQuery Mobile [13], Ionic [14] and Sencha Ext JS [15].

Two strategies can be applied to distribute the applications to end-users. First, an application (i.e., **Web app**) can be hosted on a Web server. This flexible approach allows the user to access the application in a mobile browser in a platform-agnostic manner. Hence, a large market can be easily reached and the time-to-market can be short. However, the application is constrained to the resources available in the browser. For instance, the JavaScript API of the browser (e.g., access to sensors such as accelerometer and GPS) is more limited than using native approaches. Furthermore, accessing the application is more cumbersome than starting an application installed on the device.

A second strategy is to pack the Web app into a standalone application by using a **Web-to-native wrapper**. The resulting application is often called a hybrid app due to the fact that it has both native as well as web characteristics. The packaging results in an application that can be submitted to the app stores of the different platforms. The Web app does not longer reside in the browser, but in a chromeless (without the window decoration of a regular web browser) WebView. The application consists of the WebView wrapper and the applications HTML, CSS and JavaScript files. The Web-to-native wrapper also features a JavaScript bridge that allows access to a broader range of platform APIs compared to the browser-exposed functionality. A typical structure of a hybrid application is illustrated in Figure 1. The most popular Web-to-native wrapper is Apache Cordova, formerly PhoneGap.

B. Source-Code Translators and Runtimes

Another CPT strategy is translating source code to code which can be understood by the underlaying platform such as a runtime. **Runtimes** shield applications from underlying platform differences through compatibility layers. The application source code can be either compiled or interpreted by the runtime during execution.

Also, the source code can be translated to the platform's native language or to executable byte code via a **Source-Code Translator**. Popular source code translators are Xamarin [16] and Qt [17]. In most cases, a combination of a source-code translator and a runtime is employed where a compilation step translates the source code to a binary or intermediary language that runs on the runtime. In a minority of tools, the source code will run straight on the runtime (e.g., Titanium [18]) or will be translated to native source code without a runtime (e.g., NeoMAD [19]). For each platform, the resulting source code is compiled using the development tools provided by the platform developer.

IV. Cordova Framework

A typical Cordova application consists of three important components: the application source, the WebView and plugins, as depicted in Figure 1. The Cordova framework allows these components to exchange information. A crucial component is the Cordova bridge which provides a way to connect the client-side native code with the JavaScript source.

A. Cordova Application Structure

The application code is loaded in a chromeless WebView. By default, Cordova applications use the WebView bundled

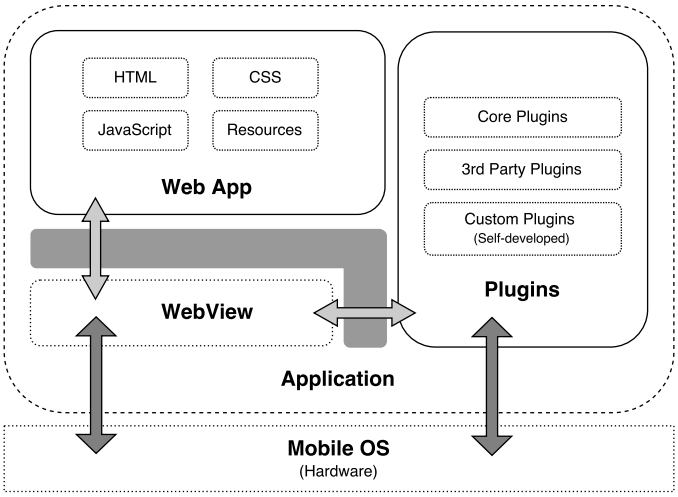


Figure 1. Structure of a Cordova application. Light grey arrows represent JavaScript calls, darker grey arrows represent native calls. The Cordova framework is illustrated by the grey area.

with the operating system. An alternative is to include the **Crosswalk WebView** [20]. The Crosswalk WebView provides uniform behaviour and interfaces between different (versions of) operating systems. Hence, developers do not need to take into account the differences in behaviour/APIs between the WebViews contained in the different (versions of) operating systems. Despite the efforts of the Crosswalk project, the maintenance and further developments have halted. The Android platform WebView has moved to an application so it can be updated separately to the Android platform. As a consequence, the crosswalk WebView will not yield a better performance on devices were the WebView is kept up-to-date. Therefore, we do not consider the impact of the crosswalk project on the performance of data storage strategies in this paper.

Cordova developers have two options for accessing device resources: the HTML5 APIs provided by the WebView and plugins. Despite the continuously growing HTML5 functionality [21] and the introduction of Progressive Web Apps [22], the JavaScript APIs provided by the WebView are not –yet– sufficient for the majority of applications. They do not provide full access to the diverse resources of the mobile device, such as sensors (e.g., accelerometer, gyroscope) and functionality provided by other applications installed on the device (e.g., contacts, maps, Facebook login). Plugins allow JavaScript code to access native APIs by using a JavaScript bridge between the Web code and the underlying operating system. Commonly used functionality such as GPS are provided by Cordova as *core plugins*. Additional functionality is provided by over 1000 third-party plugins, which are freely available in the Cordova plugin store [23]. Plugins consist of both JavaScript code and native code (i.e., Java for Android, Objective-C and recently Swift for iOS). The JavaScript code provides the interface to the developer. The native source code implements the functionality of the plugin and is compiled when building the application. The Cordova framework provides the JavaScript bridge that enables communication between JavaScript and native components. For each platform, Cordova supports several bridging mechanisms. At runtime, Cordova selects a bridging mechanism. When an error occurs, it switches to another mechanism. Independently of the selected bridging mechanism, the data requires several conversion steps before and after crossing the bridge.

B. Cordova Bridges

It is crucial to understand the inner workings of this bridge to be able to correctly evaluate the performance of the Cordova storage APIs. Here we consider the used bridge techniques on the Android and iOS platform.

Default in Android, Cordova uses the `addJavaScriptInterface` method for reaching the native side. This method injects a supplied object into the WebView. Afterwards an interface to the client-side Android code is accessible in JavaScript running in the WebView. This results in a bridge which can be invoked in JavaScript. Cordova invokes the `evaluateJavascript` method to execute JavaScript code in client-side Android.

The following bridge techniques are used in Android to exchange data and commands:

- JavaScript to Native Android:
  - **JS Object** (default). Methods of a Java object are directly accessed in Javascript via the `addJavaScriptInterface` as described above.
  - **Prompt**. The data is communicated through the prompt functionality of the WebView. This is used pre-JellyBean (i.e., Android 4.1), where `addJavascriptInterface` is disabled.
- Native Android to Javascript:
  - **Evaluation Bridge** (default). Through `evaluateJavascript` native Android code can execute JavaScript directly. This bridge was recently added and is now the default bridge.
  - **Polling**. The JavaScript side can be accessed by periodically polling for messages using the Javascript to Native Bridge.
  - **Event interception**. The Javascript code intercepts events (i.e., Load URL and Online event) and extracts the message from that event.

In iOS similar bridge techniques can be used. The javascript-to-native bridge is realized via URL loading interposition. On the JavaScript side a URL is loaded in an iframe or via XMLHttpRequest (XHR). This loading is intercepted by the native side. The JavaScript side communicates via the native side by encoding messages inside these URLs. The native side can access the JavaScript side by executing JavaScript code via the WebView's method `StringByEvaluatingJavaScriptFromString`.

V. Data Storage In Cordova

This work focuses on data storage mechanisms in Cordova applications. Four types of data storage strategies are distinguished: files, databases, persistent variables and sensitive data. Databases are used to store multiple objects of the same structure. Besides data storage, databases also provide methods to conveniently search and manipulate records. File storage can be used to store a diverse set of information such as audio, video and binary data. Persistent variables are stored as key-value pairs. It is often used to store settings and preferences. Sensitive

data (e.g., passwords, keys, certificates) are typically handled separately from other types of data. Mobile operating systems provide dedicated mechanisms that increase the security of sensitive data storage.

The remainder of this section discusses the storage APIs available in Cordova and native Android/iOS. A summary of the results is shown in Table I.

TABLE I. STORAGE API COVERAGE

	Cordova	Android	iOS
Databases	WebSQL IndexedDB SQLite Plugin	SQLite	SQLite
Files	Cordova File Plugin	java.io	NSData
Persistent Variables	LocalStorage	Shared Prefs	NSUserDefaults Property Lists
Sensitive Data	SecureStorage Plugin	KeyStore Keychain	Keychain

A. Databases

Android and iOS provide a native interface for the **SQLite** library. Cordova supports several mechanisms to access database functionality from the application. First, the developer can use the database interface provided by the WebView. Both the native and CrossWalk WebViews provide two types of database APIs: **WebSQL** and **IndexedDB**. Although WebSQL is still commonly used, it is officially deprecated and thus no longer actively supported [24]. Second, developers can access the native database APIs via the **SQLite Plugin** [25].

B. Files

In Android, the file storage API is provided by the **java.io** package, in iOS this is included in **NSData**. Cordova provides a core plugin for File operation, namely **Cordova File Plugin** (cordova-plugin-file) [26]. Files are referenced via URLs which support using platform-independent references such as *application_folder*.

C. Persistent Variables

In Android, storing and accessing persistent variables is supported via **SharedPreferences**. It allows developers to store primitive data types (e.g., booleans, integers, strings). iOS developers have two options to store persistent variables: **NSUserDefaults** and **Property Lists**. NSUserDefaults has a similar behaviour to SharedPreferences in Android. Property Lists offer more flexibility by allowing storage of more complex data structures and specification of the storage location. Cordova applications can use the **LocalStorage** API provided by the Android and iOS WebView. Although it provides a simple API, developers should be aware of several disadvantages. First, LocalStorage only supports storage of strings. More complex data structures need to be serialized and deserialized by the developer. Second, LocalStorage is known [27] to perform poorly on large data sets and has a maximum storage capacity of 5MB.

```
1 // coarse grained API
2 NativeStorage.setItem("reference_to_value",<value>,
      <success-callback>, <error-callback>);
3 NativeStorage.getItem("reference_to_value",<success-
      callback>, <error-callback>);
4 NativeStorage.remove("reference_to_value",<success-
      callback>, <error-callback>);
5 NativeStorage.clear(<success-callback>, <error-
      callback>);
```

Listing 1. NativeStorage – Coarse-grained API

```
1 // fine grained API
2 NativeStorage.put<type>("reference_to_value",<value>,
      <success-callback>, <error-callback>);
3 NativeStorage.get<type>("reference_to_value",<
      success-callback>, <error-callback>);
4 NativeStorage.remove("reference_to_value",<success-
      callback>, <error-callback>);
```

Listing 2. NativeStorage – Fine-grained API

D. Sensitive Data

Android provides two mechanisms to store credentials: the **KeyChain** and the **KeyStore**. A KeyStore is bound to one specific application. Applications can not access credentials in KeyStores bound to other applications. If credentials need to be shared between applications, the KeyChain should be used. The user is asked for permission when an application attempts to access credentials in the KeyChain. Credential storage on iOS is provided by the **Keychain**. Credentials added to the Keychain are, by default, app private, but can be shared between applications from the same publisher. Cordova developers can use the credential storage mechanisms provided by Android and iOS via the **SecureStorage** (cordova-plugin-secure-storage) [28] plugin.

VI. NATIVESTORAGE PLUGIN

An important limitation of using HTML5 APIs (e.g., IndexedDB and LocalStorage) to store data in Cordova applications is that both on Android and iOS the cache of the WebView can be cleared when, for instance, the system is low on memory. This section presents NativeStorage [29], a Cordova plugin for persistent data object storage, mitigating the limitations of the HTML5 storage mechanisms.

A. Plugin Requirements

The requirements of the plugin are listed below:

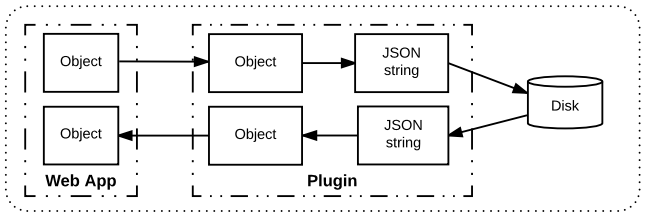
- R_1 Persistent and sufficient storage
- R_2 Storage of both primitive data types and objects
- R_3 Support for Android and iOS
- R_4 App private storage
- R_5 Responsive APIs
- R_6 A user-friendly API

B. Realisation of NativeStorage

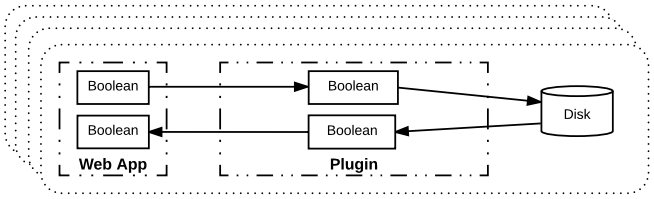
The plugin consists of JavaScript and native code. The JavaScript API provides the interface to application developers. The native side handles the storage of variables using native platform APIs.

NativeStorage provides two sets of JavaScript APIs, a fine-grained and a coarse-grained API, which are both asynchronous

and non-blocking. The coarse grained API (Figure 2a) provides a type-independent interface, variables are automatically converted to JSON objects via the JSON interfaces provided by the WebView and passed as string variables to the native side. When a value is retrieved, the WebView is used to convert the string back to an object. The fine-grained API (Figure 2b) provides a separate implementation for the different JavaScript types. On the native side, the variables are stored via SharedPreferences in Android and NSUserDefaults in iOS.



(a) Coarse-grained API



(b) Fine-grained API

Figure 2. NativeStorage API

C. Evaluation of NativeStorage

The plugin is evaluated based on the previously listed requirements.

Persistent storage is provided via the native storage mechanisms. The documentation of the used native mechanisms does not state a limitation on the storage capacity. Hence, as opposed to LocalStorage, the storage capacity is only limited by the available memory on the device, satisfying R_1.

The native part of the plugin is developed for both Android and iOS. These mobile operating systems have a combined market share of 99% [30]. The used native storage mechanisms were introduced in iOS 2.0 and Android 1.0. The plugin, hence, provides support for virtually all versions of these platforms used in practice, satisfying R_3.

The plugin uses NSUserdefaults and SharedPreferences to store the data in app-private locations, ensuring that the variables can not be accessed from outside the application. This satisfies R_4.

The APIs are implemented using an asynchronous non-blocking strategy, facilitating the development of responsive applications (conform R_5).

Web developers are familiar with duck typing used in languages such as JavaScript. These types of languages often have APIs that don't distinguish between data types. The coarse-grained API provides such a storage mechanism. This API is shown in Listing 6. Not all Cordova developers, however, have a Web background. Therefore, a fine-grained API (Listing 5) is provided for developers who are more comfortable with a statically typed language, satisfying R_6 and R_2. Using both the coarse- and fine-grained API, the different JavaScript data types can be stored. Developers, however, need to be aware that the object storage relies on the JSON interface of the WebView to convert the object to a JSON string representation. The WebView, for instance, does not support the conversion of circular data structures. These types of objects, hence, need to be serialized by the developer before they can be stored.

Since its release to Github [31] and NPM [32] the plugin has been adopted by many Cordova application developers. We have registered over 16 500 downloads per month; with an overall number of downloads of 172 250 over a time span of two years.[1] Furthermore, the plugin is part of the 4% most downloaded packages on NPM. The plugin has been adopted in Ionic Native (Ionic 2) [33] and the Telerik plugin marketplace [34]. Telerik verifies that plugins are maintained and documented, thereby ensuring a certain quality.

VII. Evaluation

The evaluation of the data storage mechanisms consists of two parts: a quantitative performance analysis and a security evaluation.

A. Performance

Developers want to be aware of the potential performance impact of using a CPT for mobile app development [12]. This section evaluates the performance of the different storage mechanisms for Cordova applications and compares the results with the native alternatives. Each storage strategy is tested by deploying a simple native and Cordova test application that intensively uses the selected storage strategy on an Android and iOS device. For Android the Nexus 6 running Android 6 was used, for iOS the IPhone 6 running iOS 9 was used. The test application communicates the test results via timing logs that are captured via Xcode for iOS and Android Studio for Android. The experiments were run sufficient times to ensure the measurements adequately reflect the performance of the tested storage mechanisms.

1) Databases

a) Test Application The database test application executes 300 basic CRUD operations (i.e., 100 x create, 100 x read, 50 x delete and 50 x read) of objects containing two string variables. The performance is determined by means of measuring the total duration of all the transactions. This test has been executed using the SQLite (native and Cordova), WebSQL (Cordova) and IndexedDB (Cordova) mechanisms.

b) Results and Comparison The results are presented in Table II. The mechanism for retrieving values by means of an index clearly results in a better performance compared to the SQL-based mechanisms. This analysis shows that IndexedDB provides an efficient way of storing and retrieving small objects. WebSQL –provided by the WebView– acts as a wrapper around SQLite. This is illustrated by the performance overhead associated with this mechanism. The deprecation of the specification/development stop could also have contributed to the performance penalty. The SQLite plugin suffers from a performance overhead caused by the interposition of the Cordova framework/bridge and has consequently a noticeable

[1] The statistics of the NativeStorage plugin can be found at https://npm-stat.com/charts.html?package=cordova-plugin-nativestorage.

performance overhead. The performance overhead introduced by the Cordova bridge is discussed in more detail in the following section.

TABLE II. RATIO OF DATABASE EXECUTION TIME TO THE NATIVE (SQLITE) OPERATION DURATION (IN %). *IN IOS INDEXEDDB IS ONLY SUPPORTED AS OF IOS 10.

	Android *Nexus 6*	iOS *iPhone 6*
SQLite (Native)	100	100
IndexedDB	6.94	12.47*
WebSQL	153	128
SQLite Plugin	133	116

2) Files

a) Test Application The test application distinguishes between read and write operations. Each operation is tested using different file sizes, ranging from small files ($\sim 1\,\mathrm{kB}$) to larger files ($\sim 10\,\mathrm{MB}$). The performance of small files provides a baseline for file access. The performance of the read and write operations itself can be determined via the results of the large files. This test has been conducted ten times for each file size. The read and write operations consist of different steps on Android and iOS. Both the duration of the individual steps and the entire operation (i.e., read/write) is measured via timestamps. The application's memory footprint is measured via Instruments tool (Activity Monitor) in Xcode and via Memory Monitor in Android Studio.

b) Results and Comparison The results of the timing analysis on Android and iOS are presented in Figures 3 and 4, respectively. In both Android and iOS a significant performance difference between the native and the Cordova mechanism can be observed. R/W operations via the file plugin take longer compared to the native mechanisms. On top of a performance overhead, Cordova also comes with a higher memory consumption, especially in iOS (Figure 5).

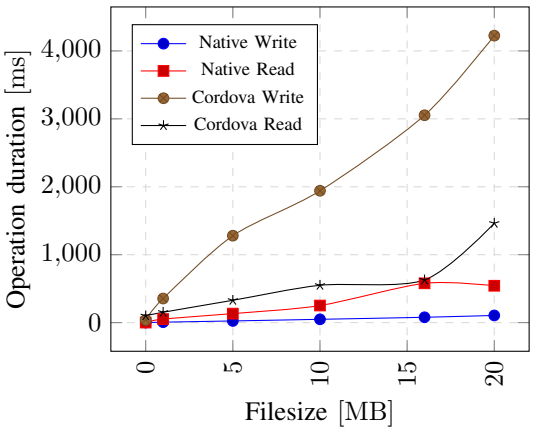


Figure 3. Duration of file operations in Android

Speed. Tables III and IV give a fine-grained overview of the different operations executed during respectively a file read and write using the Cordova platform on Android. Tables V and VI provide the results for iOS. Before data can be sent over the Cordova bridge, it needs to be converted to a string. This can create significant overhead when large binary files such as images need to be manipulated. Before they are sent over the bridge, the binary data is converted to a Base64 string. On Android, this is illustrated in the *Processing file* component of Table IV. Sending the data over the bridge also comprises a significant part of the overhead (i.e., *Sending over bridge*, from Table III). For small files, the overhead originates for the most part from resolving the platform-independent URL to a local path and retrieving meta-data. Similar observations can be made based on the iOS results.

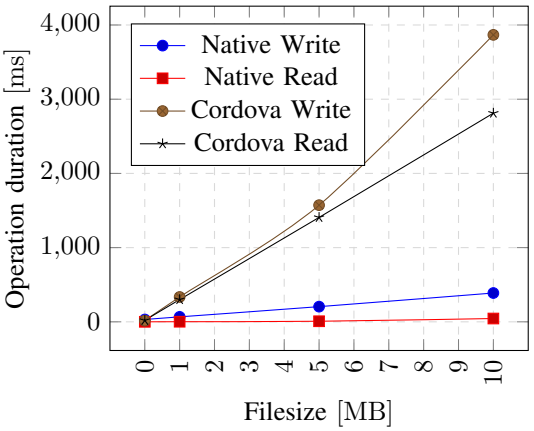


Figure 4. Duration of file operations in iOS

TABLE III. EXECUTION TIME OF COMPONENTS ASSOCIATED WITH A READ OPERATION IN CORDOVA ANDROID (FILE PLUGIN). THE PROCEDURE "SENDING OVER BRIDGE" CONSISTS OF ENCODING, SENDING AND DECODING MESSAGES FROM THE JAVASCRIPT SIDE TO THE NATIVE SIDE.

Component	**Duration** [1 MB] (ms)	(% total)	**Duration** [20 MB] (ms)	(% total)
Resolve to local URL	58	**46**	59	7.56
Native reading	20	16	366	**47**
Sending over bridge	28	22	339	**43**
Total	126		780	

Memory. In iOS, applications manipulating large files will require large amounts of memory. This is illustrated in Figure 5. As shown, reading and writing a 10 MB file results in 400 MB of allocated memory. Reading and writing files larger than 10 MB can result in unstable behavior on iOS due to the large memory requirements. A solution for developers is to split large file operations in different steps.

c) Conclusion File storage on Apache Cordova comes with a number of limitations in terms of performance. This is a result of the Cordova framework/bridge technology. Allowing binary data to pass over the Cordova bridge could significantly improve the performance of plugins that perform operations on binary data. For instance, in [35] a bridging technology is presented that allows access to native device APIs in HTML5

TABLE IV. EXECUTION TIME OF COMPONENTS ASSOCIATED WITH A WRITE OPERATION IN CORDOVA ANDROID (FILE PLUGIN). THE PROCEDURE "PROCESSING FILE" CONVERTS THE BYTES –AS AN ARRAYBUFFER– TO A STRING ARRAY. THE "EXECUTE CALL DELAY" REPRESENTS THE DELAY BETWEEN THE WRITE COMMAND EXECUTED IN JAVASCRIPT AND THE EXECUTION AT THE NATIVE SIDE.

Component	**Duration** [1 MB]		**Duration** [20 MB]	
	(ms)	(% total)	(ms)	(% total)
Processing file	108	**65**	1290	**56**
Execute call delay	38	**23**	632	**28**
Writing	20	12	369	16
Total	166		2291	

TABLE V. PERFORMANCE READ COMPONENTS IN CORDOVA IOS

Component	**Duration** [1 MB]		**Duration** [10 MB]	
	(ms)	(% total)	(ms)	(% total)
Resolve to local URL	11	3.56	16	0.6
Native reading	13.98	4.52	70	2.47
Arguments to JSONArray	202.77	**65.62**	2037.93	**71.88**
Sending over bridge	59.93	**19.39**	587.19	**20.71**
Total	309		2835	

TABLE VI. PERFORMANCE WRITE COMPONENTS IN CORDOVA IOS

Component	**Duration** [1 MB]		**Duration** [10 MB]	
	(ms)	(% total)	(ms)	(% total)
Processing file	266	**97**	2614	**96**
Native writing	7	3	96	4
Total	273		2710	

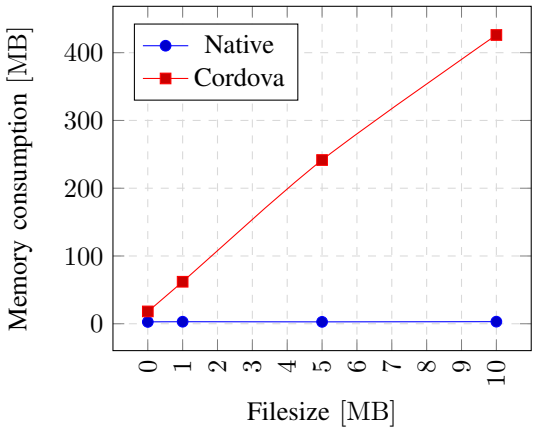


Figure 5. Memory consumption as a result of file operations in iOS

applications via WebSockets and HTTP servers, supporting the use of binary data.

3) Persistent variables

a) Test Application The performance is examined via storing and retrieving string values. The total duration of storing and retrieving a thousand variables is measured. The average storage and retrieval time is used to compare the different storage mechanisms. The Cordova mechanisms are LocalStorage and NativeStorage. These are compared to NSUserDefaults (iOS), Property Lists (iOS) and SharedPreferences (Android).

b) Results and Comparison All mechanisms have an execution time under 1 ms, with the exception of NativeStorage and Property Lists. The set operation takes around 1.9 ms, the get operation takes less than 1 ms. NativeStorage is the only mechanism which uses the Cordova bridge and framework, introducing a certain overhead. However, the NativeStorage API is asynchronous, hence, developers can continue processing while the value is being stored. The listed measurements include the time until the callback is fired. Property Lists load an entire file in an array, after which individual parameters can be read. As a consequence, the performance of the get operation, which takes 9.83 ms, is worse compared to the native alternatives. SharedPreferences and NSUserDefaults also load all parameters in memory, but this is done during the initialisation phase of the application, which is not incorporated in the measurements.

4) Cordova Bridges In addition to the evaluation of the data storage strategies in Cordova, compared to native, the performance of the utilized bridges are studied.

a) Test Application The performance of the bridges is tested by means of measuring the execution time of reading and writing different file sizes for each employed bridge. The used file sizes range from small files (1 kB) to larger files (10 MB). The test application reads and writes each file 1000 consecutively. The execution time for each transaction (i.e., read and write) is averaged over 1000 iterations. For larger files, such as $\sim$ 1 MB and $\sim$ 10 MB, the number of iterations is reduced to 100 and 20, respectively. We only consider the Android platform because the bridge techniques in iOS are rather limited.

First, the Cordova bridge allowing communication from the native to the JavaScript side (i.e., evaluation, online event, URL loading and polling) is studied. The bridge between JavaScript to native is unaltered, i.e., the default bridge is used. Second, the performance of the bridges connecting the JavaScript to the native side (i.e., JS object and prompt) is considered. For more information on the bridge mechanisms see Section VII-A4

The bridge experiments were conducted on a Samsung Galaxy S8 running Android 7.0.

b) Results and Conclusion The results [36], concerning the bridge connecting native to JavaScript, are presented in Tables VII and VIII. A first observation is that the polling mechanism could not be used as a bridge for the used Android version. In addition, the default bridge, denoted *Eval*, does not always surpass the other available bridges in terms of performance. This can be seen for reading as well as writing files. Depending on the file size the performance of other bridges are better than the default bridge. The same conclusion can be made when considering the bridges between the JavaScript side to the native side. However, the performance difference between these bridges is not as pronounced as with the native to JavaScript bridges. These results are added to this paper for

the sake of completeness. They are presented in Table IX.

Based on the measurements, we can conclude that a single bridge technique is not appropriate for every use case when considering the performance. Hence, we propose a system were plugin developers can request a certain bridge technique based on their requirements. A fine-grained flexible selection of bridges will allow the performance of the application to increase. The impact on the performance when switching between bridges is not studied in this paper.

TABLE VII. PERFORMANCE NATIVE TO JAVASCRIPT BRIDGES IN ANDROID WHEN READING FILES

Data Size [bytes]	Eval (default) [ms]	Online Event [ms]	URL Load [ms]	Polling [ms]
1k	**2.31**	6.46	3.28	N.A.
10k	**6.22**	6.80	6.74	N.A.
100k	19.60	**8.20**	32.06	N.A.
1M	106.08	**43.42**	176.71	N.A.
10M	1308.33	**641.48**	1823	N.A.

Bold measurements indicate the best performance.

TABLE VIII. PERFORMANCE NATIVE TO JAVASCRIPT BRIDGES IN ANDROID WHEN WRITING FILES

Data Size [bytes]	Eval (default) [ms]	Online Event [ms]	URL Load [ms]	Polling [ms]
1k	**19.34**	43.03	23.34	N.A.
10k	36.74	42.85	**26.20**	N.A.
100k	42.60	40.74	**37.61**	N.A.
1M	424.65	419.67	**398.50**	N.A.
10M	4465.48	5996.29	**4398.43**	N.A.

Bold measurements indicate the best performance.

TABLE IX. PERFORMANCE JAVASCRIPT TO NATIVE BRIDGES IN ANDROID WHEN READING AND WRITING FILES

Data Size	Write		Read	
[bytes]	JS Object (default) [ms]	Prompt [ms]	JS Object (default) [ms]	Prompt [ms]
1k	21.88	22.02	2.73	3.17
10k	24.01	45.82	4.36	9.12
100k	46.19	43.51	21.38	19.80
1M	403.58	431.39	107.63	99.02
10M	5679.57	5606.48	1408.86	1406.48

B. Security

On both Android and iOS the security of storage mechanisms strongly depends on the storage location and the platform's backup mechanisms. Data stored inside the sandbox of the application is only accessible by the application. However, the backup mechanisms used in iOS and Android can result in the exposure of sensitive data [37, 38, 39], or potentially exhausting the limited cloud storage capacity. On iOS, this can result in the rejection of the application (conform the Data Storage Guidelines [40]). On Android, data stored inside the application sandbox (e.g., the WebView's storage) is included if a backup is taken. The Backup API of Android can be used to explicitly blacklist data that should not be backed up. On iOS, whether or not a file is included in the backup depends on the folder in which it is stored. For instance, by default, Cordova stores the WebView's data in a folder that allows backups. This behavior can, however, be changed by modifying a Cordova parameter.

1) Databases All database mechanisms are by default private to the application and can be backed up on both mobile platforms, with the exception of the SQLite plugin in iOS. The plugin initially followed the default behaviour, but as a security measure the default storage location of the plugin in iOS was changed to a directory which is not backed up. This SQLite plugin also has an encrypted alternative, i.e., **cordova-sqlcipher-adapter**. This alternative provides a native interface to SQLCipher, encrypting SQLite databases via a user-supplied password.

2) Files In iOS files are protected by a protection class. Each of these classes corresponds to different security properties. As of iOS 7, all files are by default encrypted individually until first user authentication. The file plugin doesn't allow changing this default behaviour. Native, each file can be secured using a protection class best suited for the security requirements of that file. The plugin allows the developer to choose between folders that are public/private and backup-enabled/disabled. However, on Android backup-disabled locations can be accessed by other applications.

3) Persistent variables All persistent variable storage mechanisms are private to the application and included in backups on both mobile platforms, with the exception of Property List. Property lists can be stored in arbitrary locations, and can be backed up depending on the specified location.

4) Sensitive Data The Secure Storage plugin provides storage of sensitive data on Android and iOS. On iOS, the plugin uses the SAMKeychain [41] plugin which provides an API for the native iOS Keychain. The plugin allows app-global static configuration of the KeyChain items' accessibility. This could entail a security risk, as it does not allow fine-grained protection of individual items. When a user backs up iPhone data, the Keychain data is backed up but the secrets in the Keychain remain encrypted with a phone-specific key in the backup. The Android KeyChain only allows storage of private keys. Hence, for storing other tokens such as passwords or JWT tokens, an additional encryption layer is used. The plugin generates a key that is stored in the KeyChain and used to encrypt/decrypt sensitive data. The KeyChain on Android is not included in backups.

VIII. CONCLUSION AND FUTURE WORK

This paper presented an assessment of data storage strategies using the mobile cross-platform tool Cordova. An in-depth analysis was performed on the API coverage of the available data storage mechanisms in Cordova and Native applications. Based on the analysis, an additional Cordova storage plugin was developed that improves the storage of persistent variables.

Furthermore, the performance and security of the available storage mechanisms were evaluated. Our performance analysis shows that using the Cordova bridge comes with a significant

performance penalty. Hence, the WebView's JavaScript API should be used when possible. Moreover, we also demonstrated that the default bridges used in Cordova do not always outperform the non-default bridges. However, apart from performance, other parameters such as functionality and security can have an impact on the selection of the storage mechanism.

Databases. If access to a full fledged SQL database is required, the SQLite plugin should be used. However, in most mobile applications, the functionality provided by the significantly faster IndexedDB interface of the WebView is sufficient.

Variables. As described in Sections VI and VII, it is recommended to use NativeStorage for storing persistent variables, since LocalStorage does not guarantee persistence over longer periods of time. This type of storage is often used to store preferences. Preferences are typically only accessed once or twice during the life cycle of the application. Hence, the performance overhead of NativeStorage does not have a significant impact on the performance of the application.

Files. The WebView does not provide a file storage API. Hence, developers have to use the core plugin, Cordova File Plugin (`cordova-plugin-file`).

Sensitive data. The security analysis presented in Section VII-B shows that plugins such as SecureStorage offer increased security compared to the WebView's JavaScript API because they benefit from the platform's native secure storage APIs. It is therefore recommended to use a plugin such as SecureStorage to store sensitive data.

Future work on this topic can include an enhancement of the Cordova framework where a fine-grained selection of bridge techniques is allowed. Thereby, improving the performance of Cordova applications. Furthermore, more CPTs can be included in the assessment of data storage strategies.

References

[1] G. Callebaut, M. Willocx, J. Vossaert, V. Naessens, and L. D. Strycker, "Assessment of data storage strategies using the mobile cross-platform tool cordova," in *MOBILITY 2017, The Seventh International Conference on Mobile Services, Resources, and Users*, J. Noll and K. El-Khatib, Eds., 2017, pp. 25–32.

[2] VisionMobile. Cross–Platform Tools 2015. Access date: 13/04/2016. [Online]. Available: http://www.visionmobile.com/product/cross-platform-tools-2015/

[3] ——. (2016) Developer economics state of the developer nation q1 2016. Access date: 13/04/2016. [Online]. Available: http://www.visionmobile.com/product/developer-economics-state-of-developer-nation-q1-2016/

[4] F. Rösler, A. Nitze, and A. Schmietendorf, "Towards a mobile application performance benchmark," in *International Conference on Internet and Web Applications and Services*, vol. 9, 2014, pp. 55–59.

[5] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. IEEE, 2013, pp. 323–328.

[6] M. Willocx, J. Vossaert, and V. Naessens, "Comparing performance parameters of mobile app development strategies," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 2016, pp. 38–47.

[7] M. Ciman and O. Gaggi, "Evaluating impact of cross-platform frameworks in energy consumption of mobile applications." in *WEBIST (1)*, 2014, pp. 423–431.

[8] Y. Zhuang, J. Baldwin, L. Antunna, Y. O. Yazir, S. Ganti, and Y. Coady, "Tradeoffs in cross platform solutions for mobile assistive technology," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*. IEEE, 2013, pp. 330–335.

[9] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," in *International Conference on Web Information Systems and Technologies*. Springer, 2012, pp. 120–138.

[10] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni, "End users' perception of hybrid mobile apps in the google play store," in *2015 IEEE International Conference on Mobile Services*. IEEE, 2015, pp. 25–32.

[11] M. Palmieri, I. Singh, and A. Cicchetti, "Comparison of cross-platform mobile development tools," in *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*. IEEE, 2012, pp. 179–186.

[12] VisionMobile, "Cross-platform developer tools 2012, bridging the worlds of mobile apps and the web," *Februar y*, 2012, access date: 13/04/2016.

[13] JQuery Mobile. A Touch-Optimized Web Framework. Access date: 15/09/2017. [Online]. Available: https://jquerymobile.com/

[14] Ionic, "Hybrid vs. Native – An introduction to cross-platform hybrid development for architects and app development leaders," Tech. Rep. [Online]. Available: https://ionicframework.com/books/hybrid-vs-native

[15] Sencha. Sencha Ext JS. Access date: 15/09/2017. [Online]. Available: https://www.sencha.com/products/extjs

[16] Xamarin – Mobile App Development & App Creation Software. Access date: 29/11/2017. [Online]. Available: www.xamarin.com

[17] Qt – Cross-Platform software development for embedded & desktop. Access date: 29/11/2017. [Online]. Available: www.qt.io

[18] Appcelerator Inc. Appcelerator. Access date: 15/09/2017. [Online]. Available: http://www.appcelerator.com/

[19] NeoMAD – Cross-platform mobile development. Access date: 15/09/2017. [Online]. Available: http://neomades.com

[20] Crosswalk – Build world class hybrid apps. Access date: 29/05/2018. [Online]. Available: https://crosswalk-project.org

[21] A. Deveria and L. Schoors. Can I use ... ? Access date: 29/05/2018. [Online]. Available: https://caniuse.com/#search=HTML5

[22] Google. Progressive Web Apps. Access date: 29/05/2018. [Online]. Available: https://developers.google.com/web/progressive-web-apps/

[23] Apache Cordova. Cordova Plugins. Access date: 29/05/2018. [Online]. Available: https://cordova.apache.org/plugins/

[24] Web SQL Database documentation. Access date: 29/05/2018. [Online]. Available: https://dev.w3.org/html5/webdatabase/

[25] SQLite Plugin NPM website. Access date: 29/05/2018.

[Online]. Available: https://www.npmjs.com/package/cordova-sqlite-storage
[26] Cordova File Plugin NPM website. Access date: 29/05/2018. [Online]. Available: https://www.npmjs.com/package/cordova-plugin-file
[27] Cordova Storage documentation. Access date: 29/05/2018. [Online]. Available: https://cordova.apache.org/docs/en/latest/cordova/storage/storage.html
[28] SecureStorage Plugin NPM website. Access date: 29/05/2018. [Online]. Available: https://www.npmjs.com/package/cordova-plugin-secure-storage
[29] G. Callebaut and A. Rajiv, "NativeStorage – A Cordova Plugin," https://github.com/TheCocoaProject/cordova-plugin-nativestorage.
[30] "Smartphone os market share, q2 2016," http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015, access date: 20/10/2016.
[31] (2016) Cordova plugin NativeStorage. [Online]. Available: https://github.com/TheCocoaProject/cordova-plugin-nativestorage
[32] NativeStorage Plugin NPM website. Access date: 29/05/2018. [Online]. Available: https://www.npmjs.com/package/cordova-plugin-nativestorage
[33] NativeStorage in the Ionic Framework documentation. Access date: 29/05/2018. [Online]. Available: http://ionicframework.com/docs/v2/native/nativestorage/
[34] Cordova Plugins in the Telerik Marketplace. Access date: 29/05/2018. [Online]. Available: http://plugins.telerik.com/cordova
[35] A. Puder, N. Tillmann, and M. Moskal, "Exposing native device apis to web apps," in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft 2014. New York, NY, USA: ACM, 2014, pp. 18–26. [Online]. Available: http://doi.acm.org/10.1145/2593902.2593908
[36] G. Callebaut. (2017) Performance results bridges in cordova (android). [Online]. Available: http://dx.doi.org/10.17632/kyxc59tfmv.1
[37] P. Teufl, T. Zefferer, and C. Stromberger, "Mobile device encryption systems," in *28th IFIP TC-11 SEC 2013 International Information Security and Privacy Conference*, 2013, pp. 203 – 216.
[38] P. Teufl, A. G. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer, "Android encryption systems," in *International Conference on Privacy & Security in Mobile Systems*, 2014, in press.
[39] P. Teufl, T. Zefferer, C. Stromberger, and C. Hechenblaikner, "ios encryption systems - deploying ios devices in security-critical environments," in *SECRYPT*, 2013, pp. 170 – 182.
[40] iOS Data Storage Guidelines. Access date: 29/05/2018. [Online]. Available: https://developer.apple.com/icloud/documentation/data-storage/index.html
[41] S. Soffes. SAMKeychain. Access date: 29/05/2018. [Online]. Available: https://github.com/soffes/SAMKeychain