Assessment of data storage strategies using the mobile cross-platform tool Cordova

Gilles Callebaut, Lieven De Strycker
firstname.lastname@kuleuven.be
DRAMCO, ESAT

Michiel Willocx, Vincent Naessens, Jan Vossaert
firstname.lastname@cs.kuleuven.be
MSEC, imec-DistriNet

***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 APIs. Apart from the performance impact, this additional layer may also result in the suboptimal use of native APIs. This paper analyses the impact of this abstraction layer using a data storage case study. Both the performance overhead and API coverage is discussed. Based on the analysis, an extension to the cross-platform storage API is proposed and implemented.**

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 (CPTs). A significant part of the code base is shared between the implementations for the multiple platforms. Moreover, many cross-platform tools such as Cordova use Web-based programming languages to implement the application logic, facilitating programmers with a Web background.

Although several cross-platform tools became more mature during the last few years, some scepticism 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 API, 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. Cordova is one of the most used CPTs [22, 23]. It is a Web-to-native wrapper, allowing the developer to bundle Web apps into standalone applications.

Contribution. The contribution of this paper is threefold. First, four types of data storage strategies are distinguished in the setting of mobile applications. The support for each strategy using both native and Cordova development is analysed and compared. Second, based on this analysis a new Cordova plugin is designed and developed that extends the Cordova Storage API coverage. 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 discusses the inner workings of Cordova applications, followed by an overview of data storage strategies and their API coverage in Cordova and native applications. The design and implementation of a new Cordova storage plugin is presented in Section IV. Section V presents a security and performance evaluation of the available Cordova and native storage mechanisms. This evaluation is followed by a general reflection. 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. [26] and Dalmasso et al. [18] evaluate the behavioral performance of cross-platform applications using parameters such as start-up time and memory consumption. Willocx et al. [30] extend this research and include more CPTs and criteria (e.g. CPU usage and battery usage) in the comparison. Further, Ciman and Gaggi [17] 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. [31] 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. [19] use criteria such as development environment, maintainability, speed/cost of development and user-perceived application performance. The user-perceived performance is analyzed further in [20], 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 [24]. 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. Data Storage In Cordova

A. Cordova Framework

A typical Cordova application consists of three important components: the application source, the WebView and plugins.

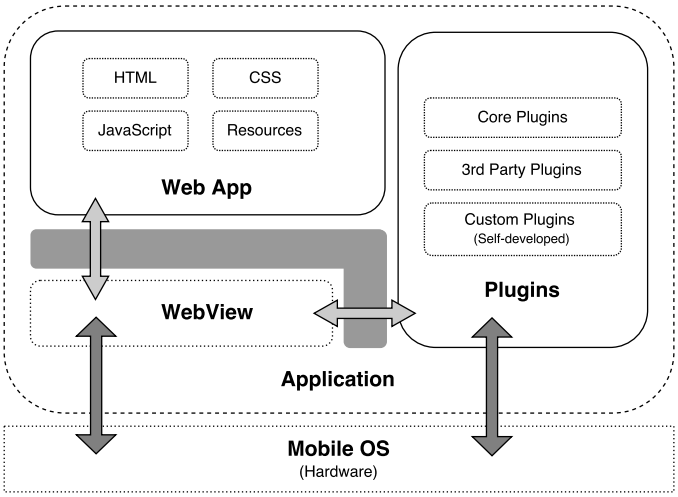


Figure 1. Structure of a Cordova application. Light grey arrows represent JavaScript calls, darker grey represent native calls. The Cordova framework is illustrated by the gray area.

Cordova applications are, similar to Web apps, developed in Web languages (i.e. HTML, CSS and JavaScript). Typically, developers use JavaScript frameworks such as Ionic and Sencha, which facilitate the development of mobile UIs.

The application code is loaded in a chromeless WebView. By default, Cordova applications use the WebView bundled with the operating system. An alternative is to include the Crosswalk WebView [13]. The Crosswalk WebView provides uniform behaviour and interfaces between different (versions of) operating systems.

Cordova developers have two options for accessing device resources: the HTML5 APIs provided by the WebView and plugins. Despite the continuously growing HTML5 functionality [11] and the introduction of Progressive Web Apps [8], 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. 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 mechansims. At runtime, Cordova selects a bridging mechanism. When an error occurs, it switches to another mechanism. Independent of the selected bridging mechanism, the data requires several conversion steps before and after crossing the bridge. 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 [12].

B. Storage API Coverage

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.

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

TABLE I. Storage API Coverage

1) 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 [10]. Second, developers can access the native database APIs via the **SQLite Plugin** [6].

2) 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) [4]. Files are referenced via URLs which support using platform-independent references such as *application_folder*.

3) 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. Furthermore, in both Android and iOS [1], the data is stored in the cache of the WebView, which can be cleared when, for instance, the system is low on memory. Last, LocalStorage is known [1] to perform poorly on large data sets and has a maximum storage capacity of 5MB.

4) 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 plugin, **SecureStorage** (cordova-plugin-secure-storage) [7].

IV. NativeStorage Plugin

Section III-B identified several limitations when using the Webview's LocalStorage to store variables in Cordova. These limitations are inconvenient for developers as they often rely on persistent storage of variables. This section presents a Cordova plugin for persistent variable storage, mitigating the limitations of the LocalStorage mechanism.

A. Requirements

The plugin tackles the main disadvantages of LocalStorage by providing:

- R_1 Persistent and sufficient storage
- R_2 Storage of both primitive data types and objects

Other requirements are:

- R_3 Support for Android and iOS
- R_4 App private storage
- R_5 Responsive APIs
- R_6 A user-friendly API

B. Realisation

The plugin consist 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.

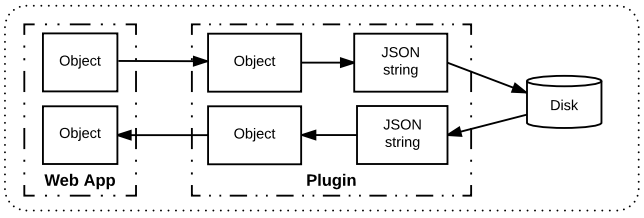
NativeStorage provides two sets of JavScript APIs, a fine-grained and a coarse-grained API, which are both asynchronous and non-blocking. The coarse grained API provides a type-independent interface, variables are automatically converted to JSON objects via the JSON interfaces provided by the WebView and passed as string variables to the native side. When a value is retrieved, the string should be converted to the desired object by the developer. The fine-grained API (Figure 2b) provides a seperate implementation for the different JavaScript types. On the native side, the variables are stored via SharedPreferences in Android and NSUserDefaults in iOS.

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

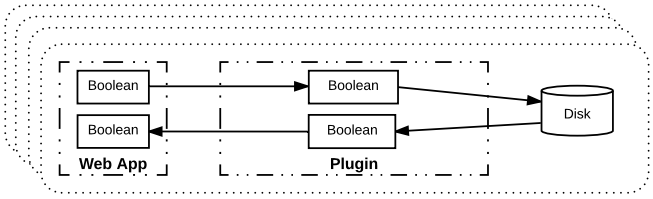
Listing 1. NativeStorage – Coarse-grained API

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

Listing 2. NativeStorage – Fine-grained API



(a) Coarse-grained API



(b) Fine-grained API

Figure 2. NativeStorage API

C. Evaluation

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 doesn't state a limitation on the storage capacity. Hence, as opposed to LocalStorage, the storage capacity is 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% [15]. The used native storage mechanisms were introducted in iOS 2.0 and Android 1.0. The plugin, hence, provides support for virtually all version 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 (cf. R_5).

Web developers are familiar with dynamic programming languages such as JavaScript, supporting type changes of objects at runtime. Thus, Web developers are more familiar with APIs that don't distinguish between different data types. The coarse-grained API provides such a storage mechanism. Not all Cordova developers have a Web background. Therefore, a fine-grained API is provided for developers who are more comfortable with a type-based mechanism, 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 strucures. These types of objects, hence, cannot be stored using the plugin.

Since its release to Github [16] and NPM [5] the plugin has been adopted by many Cordova application developers. We've registered over 2500 downloads per month. Furthermore, the plugin is part of the 5% most downloaded packages on NPM. The plugin has been adopted in Ionic Native (Ionic 2) [3] and the Telerik plugin marketplace [9]. Telerik verifies that plugins are maintained and documented, thereby ensuring a certain quality.

V. Evaluation

The evaluation of the data storage mechanisms consists out of three 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 [21]. 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 timining 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 and has consequently a noticeable performance overhead.

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

TABLE II. Ratio of database execution time to the native (SQLite) operation duration (in %). *In iOS IndexedDB is only supported as of iOS 10.

2) Files:

a) Test Application: The test application distinguishes between read and write operations. Each operation is tested using different files sizes, ranging from small files ($\sim 1\,\mathrm{kB}$) to larger files ($\sim 10\,\mathrm{MB}$). The performance of small files allows the evaluation provides a baseline for file access. The performance of the read and write operations itself can be determined via the results from the large files.

The execution duration of different procedures and components as well as the total read and write duration is measured by means of 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 are presented in Figure 3 and 4. In both Android and iOS a performance difference between the native and 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).

Android. Encoding, sending and decoding messages amounts to 43% of the total read duration for large files, but it is negligible for small files. Operations on smaller files result in a overhead originating from resolving the URL to a local path, and retrieving meta-data. When using small files, 46% of the total read duration is spent on procedures related to files access.

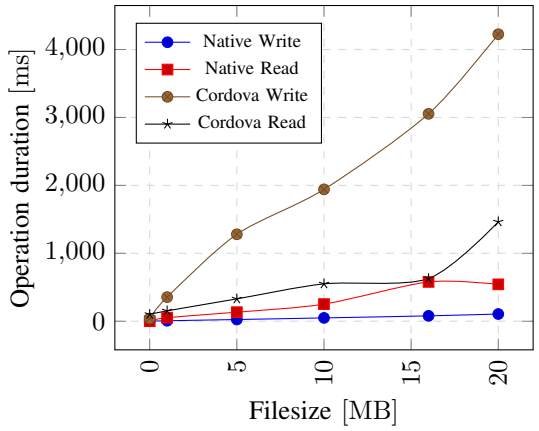


Figure 3. Duration of file operations in Android

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	

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 end decoding messages from the JavaScript side to the native side.

A larger overhead is observed when writing a binary file in Cordova. This additional overhead is created by the manner in which messages are delivered over the bridge. The file contents are stored as bytes. These bytes are first parsed to strings, after which they are sent over the bridge. Once arrived at the native side, the file contents are converted back to bytes and encoded to a Base64 format. The additional overhead originates from the procedures related to converting bytes to strings and vice versa.

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

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.

iOS. In iOS applications operating on large files will result in a high memory allocation. This is illustrated in Figure 5. Therefore, file sizes greater than 10 MB can't be read and written in Cordova. For instance, reading and writing a 10 MB file results in 400 MB of allocated memory. Another reason big files can't be transported is because of the mechanism behind the Cordova bridge, i.e. URL loading interposition. As a result, large files can't get across the bridge. A solution for developers is operating on chunks of data at a time.

The overhead as a result of reading files originates mostly from the parsing of the arguments to an intelligible format for the JavaScript code, i.e. a JSONArray. A mere 2.5% of the total operation time is spent reading file contents. Sending this content back over the bridge involves another 20% of the total duration time.

During the writing operation 97% of the total operation time is used parsing the arguments to an interpretable format by the native side; similar to Cordova Android.

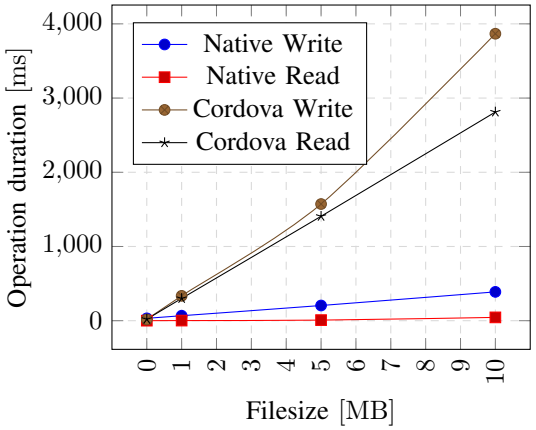


Figure 4. Duration of file operations in iOS

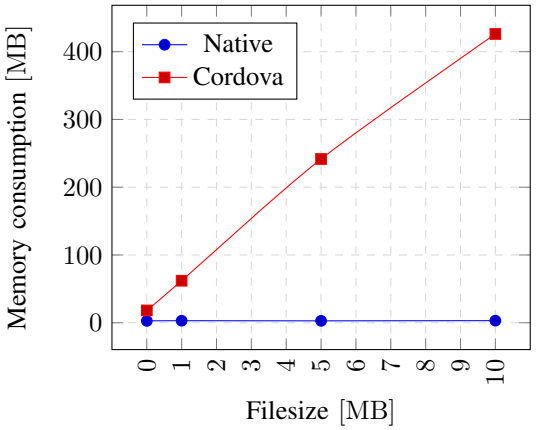


Figure 5. Memory consumption as a result of file operations in iOS

c) Conclusion: Apache Cordova is limited – performance-wise– in file operations on large files. This is a result of the used bridge technologies and the architecture of the Cordova framework. The latter is associated with a high memory consumption. Arguments (including data) need to be converted to an intelligible format to the other side, which requires time and memory.

A performance improvement could be made if the bridge allows the passing of bytes. The Cordova bridge could be, for

Component	Duration [1 MB] (ms)	(% total)	Duration [10 MB] (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 V. Performance read components in Cordova iOS

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

TABLE VI. Performance write components in Cordova iOS

instance, complemented by the *device-local service* presented in [25]. This bridge technique allows access to native device APIs in HTML5 applications via WebSockets and HTTP servers.

3) Persistent variables:

a) Test Application: The performance is examined via storing and retrieving string values. The total duration of storing and retrieving 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) and SharedPreferences (Android).

b) Results and Comparison: All mechanisms have an execution time under 1 ms, with the exception of NativeStorage in Cordova and Property Lists in iOS. NativeStorage is the only mechanism which uses the Cordova bridge and framework. In addition to the Cordova induced performance penalty, in Android the plugin's native code blocks –with each iteration– till the value is persistently stored on the disk. The plugin uses SharedPreferences in a synchronous manner to provide errors when storage to disk fails. However, SharedPreferences can be used in an asynchronous manner as utilized in the native test application. Nevertheless, the set and get operation duration in NativeStorage is negligible (i.e. 1.9 ms for storing a variable and less than 1 ms for retrieving it). Furthermore, plugin users can opt to ignore the returned success or error value –which was not the case in the test application– if they don't want to wait till the value is stored.

Property Lists in iOS are used to load an entire file in an array. The performance as a consequence of this loading can be seen in the duration of the get operation, which takes 9.83 ms. SharedPreferences and NSUserDefaults also load the entire document in memory, but this is done during the initialisation phase of the application. This phase is not incorporated in the measurements. Hence, these measurements do not reflect the total performance of the implementation, but rather the noticeable impact on the retrieval and storage of persistent variables.

B. Security

In both Android and iOS the security of storage mechanisms strongly depends on the storage location and the platform's backup mechanisms. In iOS, interactions with the file system are restricted to directories inside the application's sandbox. The Linux kernel –the centre of the Android platform– provides similar security measures to handle file system access. The backup mechanisms used in iOS and Android can result in the exposure of sensitive data [27, 29, 28]. By default, application data is backed up in Android. In Cordova Android this default behaviour is not changed. Cordova iOS allows the WebStorage to be backed up to the iCloud. This can come with two mayor implications: (1) possible leakage of sensitive data and (2) exhausting the limited iCloud storage capacity. The latter can result in the rejection of the application because the iOS Data Storage Guidelines [14] are not followed.

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 by allowing backup. However, as a security measure the default storage location of the plugin in iOS was changed to a directory which can not be 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 change this default behaviour. Natively each file can be secured using a protection class best suited for the security requirements of that file.

3) Persistent variables: All persistent variable storage mechanisms are by default private to the application and can be backed up 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 in Android and iOS.

On iOS, the plugin uses the SAMKeychain [2] plugin which facilitates manipulations on the iOS Keychain. The plugin allows static configuration of the KeyChain items' accessibility. this could entail a security risk. Data needs to be protected by the most strict data protection class. This cannot be guaranteed using a static global accessibility configuration of KeyChain items. 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.

VI. Conclusions

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. 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 IV and V, 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 de 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 V-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 might include an in-depth analysis of the CrossWalk WebView. Currently, Cordova applications suffer from a major performance penalty every time the JavaScript bridge is accessed. CrossWalk has its own plugin mechanism, which could show better performance than Cordova plugins.

References

[1] Cordova storage documentation. URL https://cordova.apache.org/docs/en/latest/cordova/storage/storage.html.

[2] Samkeychain. URL https://github.com/soffes/SSKeychain.

[3] Nativestorage in the ionic framework documentation. URL http://ionicframework.com/docs/v2/native/nativestorage/.

[4] Cordova file plugin npm website, . URL https://www.npmjs.com/package/cordova-plugin-file.

[5] Nativestorage plugin npm website, . URL https://www.npmjs.com/package/cordova-plugin-nativestorage.

[6] Sqlite plugin npm website, . URL https://www.npmjs.com/package/cordova-sqlite-storage.

[7] Securestorage plugin npm website, . URL https://www.npmjs.com/package/cordova-plugin-secure-storage.

[8] Progressive web apps. URL https://developers.google.com/web/progressive-web-apps/.

[9] Cordova plugins in the telerik marketplace. URL http://plugins.telerik.com/cordova.

[10] Web sql database documentation. URL https://dev.w3.org/html5/webdatabase/.

[11] Can i use ... ? URL http://caniuse.com/.

[12] Cordova plugins website. URL https://cordova.apache.org/plugins/.

[13] Crosswalk website. URL https://crosswalk-project.org.

[14] ios data storage guidelines. URL https://developer.apple.com/icloud/documentation/data-storage/index.html.

[15] Smartphone os market share, q2 2016. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015. access date: 20/10/2016.

[16] Cordova plugin nativestorage, 2016. URL https://github.com/TheCocoaProject/cordova-plugin-nativestorage.

[17] Matteo Ciman and Ombretta Gaggi. Evaluating impact of cross-platform frameworks in energy consumption of mobile applications.

[18] Isabelle Dalmasso, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. 2013.

[19] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating cross-platform development approaches for mobile applications. 2012.

[20] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. End users' perception of hybrid mobile apps in the google play store. In *2015 IEEE International Conference on Mobile Services*, pages 25–32. IEEE, 2015.

[21] Vision Mobile. Cross-platform developer tools 2012, bridging the worlds of mobile apps and the web, 2012. access date: 13/04/2016.

[22] Vision Mobile. Cross-platform tools 2015, 2015. URL http://www.visionmobile.com/product/cross-platform-tools-2015/. access date: 13/04/2016.

[23] Vision Mobile. Developer economics state of the developer nation q1 2016, 2016. URL http://www.visionmobile.com/product/developer-economics-state-of-developer-nation-q1-2016/. access date: 13/04/2016.

[24] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. 2012.

[25] Arno Puder, Nikolai Tillmann, and Michał Moskal. Exposing native device apis to web apps. 2014.

[26] Florian Rösler, André Nitze, and Andreas Schmietendorf. Towards a mobile application performance benchmark. 2014.

[27] Peter Teufl, Thomas Zefferer, and Christof Stromberger. Mobile device encryption systems. In *28th IFIP TC-11 SEC 2013 International Information Security and Privacy Conference*, pages 203 – 216, 2013.

[28] Peter Teufl, Thomas Zefferer, Christof Stromberger, and Christoph Hechenblaikner. ios encryption systems - deploying ios devices in security-critical environments. In *SECRYPT*, pages 170 – 182, 2013.

[29] Peter Teufl, Andreas Gregor Fitzek, Daniel Hein, Alexander Marsalek, Alexander Oprisnik, and Thomas Zefferer. Android encryption systems. In *International Conference on Privacy & Security in Mobile Systems*, 2014. in press.

[30] Michiel Willocx, Jan Vossaert, and Vincent Naessens. Comparing performance parameters of mobile app development strategies. 2016.

[31] Yanyan Zhuang, Jennifer Baldwin, Laura Antuñna, Yağız Onat Yazır, and Sudhakar Ganti. Tradeoffs in cross platform solutions for mobile assistive technology. 2013.