Differences in Inter-App Communication Between Android and iOS Systems

JCSEM December 2018

Aimun Khan Seth Lee Jiawei Wang
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas

***Abstract*— Mobile applications have evolved beyond being individual processes used to complete isolated tasks. Today, many applications operate cohesively to complete more complex tasks and offer users broad services. Individual applications behave as modules in the user's overall experience in completing a task. For example, travel apps, payment apps, and social media apps all behave seamlessly throughout a user's singular vacation experience. Inter-app communication acts as a bridge between isolated processes to provide functionality to others, and given the paradigm shift in mobile applications, this tool has become increasingly significant as a mechanism to enable these complex processes.**

This paper aims to compare inter-app communication between iOS apps to that of Android. Currently, there is much research that uses Android as a platform for testing and analysis, but comparatively little research regarding iOS as a platform. There is particularly very little research on iOS in the context of inter-app communication. Analysis of the differences between these two platforms will give us insight into the strengths and limitations of inter-application communication in different contexts, which contributes to research regarding the differences in design choices of inter-application frameworks.

I. INTRODUCTION

Third-party applications are one of the most defining features of smartphones. They enable the user to customize their smartphone experience to their particular needs, allowing the developers of mobile operating systems to focus on maintaining core operating system features instead of many pre-installed apps for all use cases. Instead of being built into the operating system, many applications today are designed and maintained by developers who can better focus on making a wide variety of apps for specific needs.

For security reasons, it is important that applications on a mobile device are isolated. Malicious applications should not be able to compromise the operating system or sensitive user data on other apps [3]. However, enabling some level of safe communication between applications is important. Apps are powerful because they are modular. Instead of being forced to start from scratch, developers are able to build off of existing functionality in other apps, creating a seamless user experience.

Inter-app communication protocols bridge this gap between allowing applications to be isolated from each other and enabling applications to share data. Applications are able to remain isolated while being able to handle specific queries from other applications, enabling them to seamlessly integrate with existing features of those other apps [10]. Inter-app communication mechanisms enable developers to allow other applications limited access to resources in their application. Android apps contain several different components that form the entire application, including activities that represent each screen with a user interface, services that run in the background, broadcast receivers that listen to system events to provide specific functionality for applications, and content provider that stores the data. These different components are usually defined inside a manifest file. Inter-app communication occurs when these components of different applications need to transfer data or open a new activity. Android implements this using intents to request other applications to carry out certain operations.

Compared to Android development, iOS applications are considerably more restricted in their permissions. Apps are viewed as isolated containers with minimal privileges [12]. These containers are referred to as sandboxes, and they require explicit permission to use services outside of the app (e.g. camera, microphone, etc). Sandboxing applications also prevent apps from writing to parts of the file system that they should not be allowed to [11]. Given iOS's emphasis on isolation of applications, URL schemes are the only way to send queries to different containers. These URL schemes somewhat have the same goal as Android intents to enable inter-app communication, but the way that this goal is accomplished by iOS is different than Android.

Currently, many papers use Android as a starting point for research on mobile development, but there is comparatively little existing research on iOS development. There is particularly little existing comparative analysis between the two mobile operating systems, especially in the context of inter-app communication. This paper aims to contrast the differences in implementation of inter-app communication in Android and iOS to explore the different design choices in the operating systems. Section II describes the implementation of inter-app communication in Android and in iOS. Section III describes the apps that we implemented to test and compare Android and iOS implementations of inter-app communication. Section IV is a summary of our findings. Section V analyzes those findings. Section VI examines threats to the validity of our findings. Section VII briefly

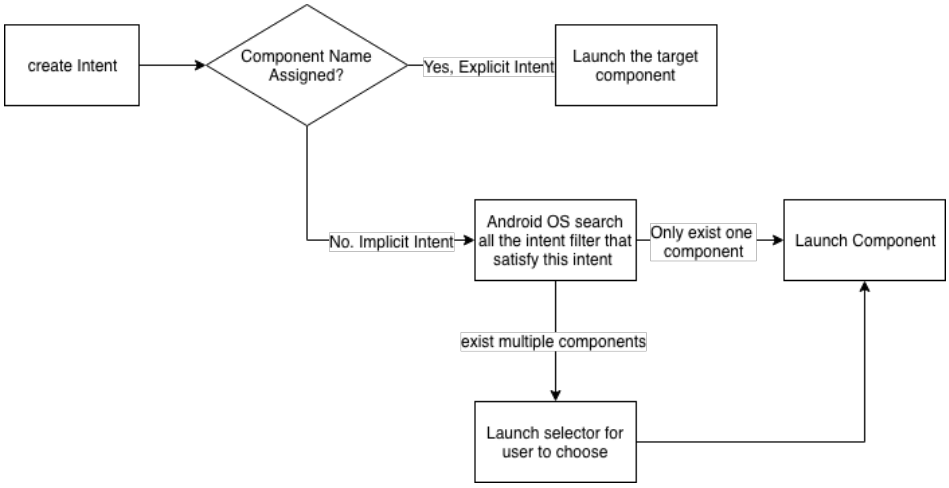

Fig. 1. Categories of intents and how operating system analyzes them

summarizes related work that we found during our research. Section VIII discusses potential areas of future research related to our findings.

II. IAC of Two Platforms

A. Implementation of IAC in Android

Each component in an Android application has specific tasks that compose the application when they communicate. The user interface, or activity, allows the user to make requests to the application to perform tasks such as playing music or viewing a picture. When a user makes a request that requires the application to transition to another interface or activity, Android applications use Intents to define this action's intent to do something. Intents allow the system to make requests to tasks both in the current app as well as other apps. Intents become the interface for different activities to communicate with each other to complete larger processes. Those processes become the interface for different applications to communicate with each other [4].

As shown in Figure 1, intents are divided into two types: implicit intents and explicit intents. Whenever developers create new intents, they have to decide whether to manually set component and activity names to this intent. If an intent is assigned to a specific component, the operating system will start the component directly. These types of intent are called explicit intents. However, if there is no specific component assigned to an intent, the Android OS will search all the applications for intent filters to determine where to send this intent. If there is only one component that accepts this type of intent, the Android system will launch that component directly. Otherwise, the system prompts the user to decide which component to send the intent to. These types of intents are called as implicit intents.

An Intent is a very common object that been used to communicate between different components. The basic instruction of an intent is to launch or deliver data to other components. As a message delivery object in Android, the most critical part of an intent is its content. Android OS will decide what to do with an intent based on what combination of the different categories of information it contains. Figure 2 shows each parameter of an intent is optional [6].

Component name is necessary information for explicit intents. When component name is set, the intent will forward directly to the designated component.

Action defines the operations that a component carries out when it receives the intent. For example, in a photo app, the main action is to display a photo. Normally, the most common intent action is the system default action. The default components can react to these action such as `ACTION_VIEW` and `ACTION_SEND`.

Data is an optional setting for an intent. It contains URI object and mime type. The former indicates the location of the pending data; the latter indicates the data type. Mime type is a very useful parameter for an implicit intent to find its best component. For instance, `ACTION_VIEW` is able to response with lots of different activities. Setting the mime type to `image/jpeg` or `video/mp4` helps to better locate the components.

Category is a string that contains the type components that are able to receive this intent.

These first four parameters mentioned above determine how Android will respond to and resolve the intent. There are two additional parameters considered optional that do not affect how Android analyzes and resolves the intent.

Extra contains information inside an intent for an action to use to complete its operation. This is the most common way for two components to communicate. Developers can easily create a *Bundle* object that contains the key-value information inside an extra to insert into an intent. Instead of passing *Bundle* objects, developers can also make their own objects `Serializable` and `Parcelable` in order to put them in intent.

Flag indicates how Android OS launches activities. For example, the flag can tell the system which task this activity belongs to and what to do after the intent is launched.

Content Provider is one of the most significant Android

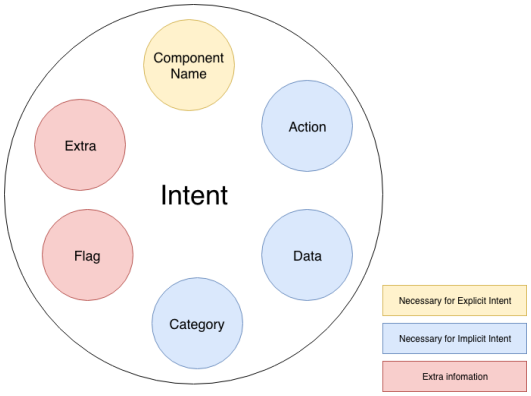

Fig. 2. Parameters inside of intents

components and is used to encapsulate data, providing it to other applications. An intent functions as a bridge to transfer data from different components, while a Content Provider is used to expose data to other apps [8]. When users want to open a certain file in a certain app, Android OS uses FileProvider, which is a subclass of Content Provider, and grants the app permission by calling the Intent.setFlags() method. These permissions are only available as long as the stack for receiving activity is active. Because the permissions are temporary, using content URI is safer than using `file://` URI. Using `file://` URI requires modifying the permissions of the underlying file and exposing these permissions to every other app, which breaks the security principle of least privilege. Thus, the increased security level offered by content URI is a key factor of Android security infrastructure.

B. Implementation of IAC in iOS

Swift app development for iOS uses a URL scheme for inter-app communication. Each app has a URL that other apps can use to make requests by passing queries to the app via the URL, which can only contain ASCII characters [9]. The URL contains argument information for a predefined URL scheme. For example, the URL `tel://1234567890` would launch the phone app and call the number 1234567890. This URL scheme is similar in functionality and implementation to explicit intents in the Android SDK. However, there is not a Swift equivalent to implicit intents. There are two types of URL schemes: built-in handlers for functions like `mailto`, `maps`, and `sms`; and custom handlers made by app developers.

With custom handlers, app developers can define a protocol to parse the query in the URL scheme. The system calls the application's openURL method to check the URL and launch the application to do a specific predefined task specified by the query, as shown in figure 3. If the app is already running in the background, it is simply moved to the foreground to open the URL, as shown in figure 3.

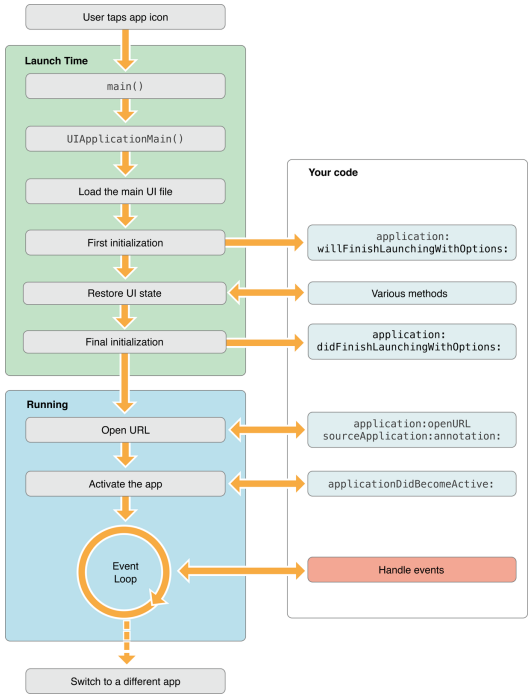

Fig. 3. Launching an app to open a URL

III. Experiments

A. Design of test apps

We designed and implemented two basic apps to communicate with each other in order to compare the Android and iOS implementations of inter-app communication[1].

Our first test app allows a user to select their favorite color out of a predetermined list of colors. Our second app displays the name of a color. Pressing a button on the first app opens the second app and changes the color to the user's favorite color. This simple design allows us to contrast the differences in implementation between Android and iOS for sending the favorite color information from app 1 to app 2. Also, by implement this simple app by ourselves, we can get a direct sense of user interface intuitiveness from the coding.

In order to compare the performance of inter-app communication in Android and iOS, we performed another test by calculating the latency in time by Android and iOS in launching from one app to another. We measured latency passing different sizes of data from one app to another.

B. Implementation of Android apps

For our Android experiment, we used Android SDK 25 to compile all our code, which ran on Android 9.0. Our applications were run on an emulator which has a Nexus 5X system Android phone with Android API version 28.

We designed two apps that used intents to pass data from the first app to the second. The first app contained buttons corresponding to various colors. Clicking a color triggers the `onClickListener()` method, which creates an Intent

[1]`https://github.com/AimunKhan/JCSEM-2018-IAC`

and sets parameters on this intent to call another app in the system.

Here we used two different implementations of intents. The first method is an explicit intent. A component name is assigned to the intent so that when intent is launched, Android OS is able to locate the activity. The second implementation is an implicit intent, which only set the action of this intent and let Android system decide which component to launch.

Next, we designed a simple test to calculate the latency time of the launch action. By printing timestamps before the intent is called and then after the second activity is created, we can get the time difference to approximate the performance of inter-app communication in the operating system. We also measured the latency by setting different sizes of data into our intent to evaluate their performance.

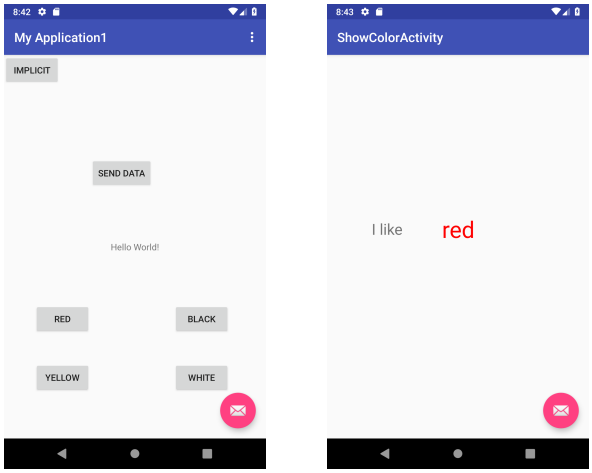

Fig. 4. When user clicks their favorite color in the first activity, the system launch another app and open the activity that receive this intent and open it

C. Implementation of iOS apps

For our Swift experiment, we used iOS version 11.2 and tested the applications on an iPhone 8 Plus emulator built into XCode.

We designed a set of iOS apps to mimic the functionality of the Android apps, making the implementation as close as possible to the Android implementation. The iOS implementation of our color apps is similar to the explicit intent implementation in Android. Pressing a button on the first app sends this data to the second app via a URL invocation. When the button is pressed by the user, the second app is called via the URL `MobileComputingApp2://\(favecolor)`, where favecolor is one of four strings chosen by user input. App 1 opens this URL, and a handler in the second app processes the query "favecolor". Based on the selected color, the RGB values and context of a displayed text message changes on open.

We also calculated the latency between pressing the button and the color change taking place using the system clock of the iPhone (iPhone 8 Plus emulator). We printed the time at button press before the URL is constructed or processed, as well as the time right after the second app is launched and rendered.

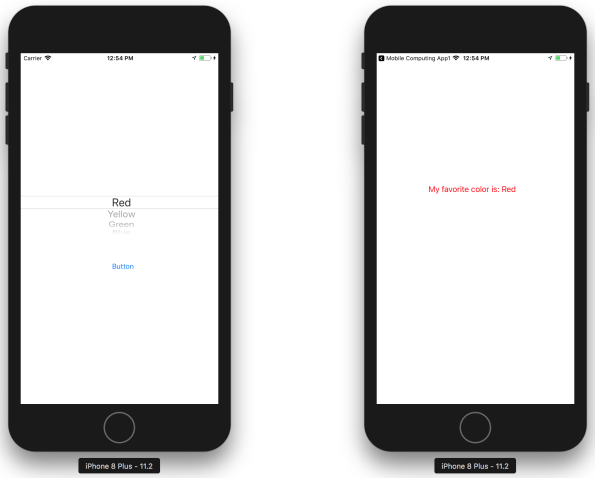

Fig. 5. The iOS apps have the same functionality as the Android apps

IV. Results

A. Observation on Android

Using explicit intents, our Android apps had a latency of 0.2340 seconds on average between the time the button was pressed on the first app and the time the second app opened and started to render, assuming the second app was not currently running. If the second app had already been launched, the latency dropped to 0.0320 seconds on average.

In contrast, using implicit intents, the Android apps had a latency of 0.4300 seconds on average when the app was not launched, and 0.0910 seconds on average when the app had already been started. This difference is intuitive and reasonable because Android OS has to spend more time on finding the right activity to start when an implicit intent is called instead an explicit intent.

We also noticed that Android's launch sequence for opening the second app changed when setting a different *flag* into the intent. For example, the app created a new task and added this new activity to the top of the history stack when the *flag* was set to `FLAG_ACTIVITY_NEW_TASK`. When we did not assign a specific FLAG to the intent, the activity would directly show in the current app and write to the current history instead.

We found some interesting results when sending different sizes of data using intents. Android has a data transfer limitation for intent. If the data you being transferred is larger than 1MB, Android will throw a `TransactionTooLargeException`. The mechanism behind this is that Android OS is trying to store the *Parcel* objects in the *Binder* transaction buffer, and this buffer is shared by all transactions in progress. Thus, comparing time consumption between different sizes of data becomes an impossible task; there is no significant difference in latency

between passing 10KB and 1MB data because there is not a significant enough size difference between these intents.

TABLE I

Latency (seconds)	Explicit Intent	Implicit Intent	URL Scheme
Initialize app	0.2340	0.4300	0.2150
Already open	0.0320	0.0910	0.0470

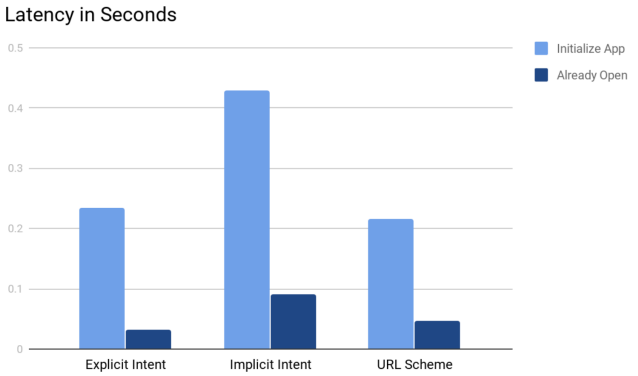

Fig. 6. URL scheme had slightly lower latency than explicit intents, while implicit intends had much higher latency than the other options

B. Observation on iOS

The iOS applications had a latency of 0.2150 seconds between the time the button was pressed on the first app and the time the second app opened and rendered. If app 2 was already open, the latency was 0.0470 seconds. This had a lower latency than the Android app.

We hypothesize that this is because iOS is more isolated and has less running background processes for each app like implicit intents, so data transfer is marginally faster. We also suspect that because iOS development is more limited in the type of data that can be sent between apps, opening a URL may be marginally more optimized for transfer speeds compared to Android intents.

In regard to iOS URLs, because data is passed via query in a URL, the size of the transfer does not scale significantly with the size of the string.

V. Analysis

A. Performance

Although the latency between apps on Android and iOS are fairly similar, iOS is marginally faster. Implicit intents are much slower than explicit intents, which makes sense given that implicit intents are more passive than explicit ones. The advantage of implicit intents is that the application does not need to know what app it is opening to handle a request, e.g. it does not know what maps app the user prefers, so it sacrifices performance for flexibility.

As for transferring big data in Android, since Android OS right now is not able to handle data that is larger than 1MB, there are typically two ways to handle this situation. The first way to do so is to write the data that developer wants to transfer inside a temporary file or a database system. When developer pass the intent, they store the path of the data. After Android 7.0, To share files between applications, developer should send a `content://` URI and also grant a temporary access permission on the URI. Content URIs with temporary UTI access permissions are secure because they apply only to the app that receives the URI, and they will expire automatically [7].

The component that accepts this intent will have to read the data from the temporary file or database. The disadvantage of this method is that Android OS will have to spend more time on reading and writing files which leads to a low efficiency. Developers can also store their data inside a static class (the component that you want to send the intent to must belong to the same process because static data is only allowed to be shared in the same process). This helps a lot when developers want to communicate inside the same app, but it can only be applied on the same process.

Swift applications have processes similar to the document controller to send files between apps, including build-in apps and processes. iOS has built-in handlers called the `UIActivityViewController` and `UIDocumentInteractionController`. These handlers provide support and manage sending of data such as pictures or documents between apps. The data is staged in this built in handler and accessed from another app via the same handler. When sharing files, iOS does not create an object for the file, but it does share a URL that points to the file. These controllers allow direct users to shared files and allow users to interact with them. Because these files are shared via built-in controllers, there are limited types of files that can be shared. However, use of these controllers increases the security of iOS inter-app file sharing by limiting the ability malicious files to be sent to other apps.

B. Security Threats to Android IACs

Apps in smartphones are regarded as isolated entities and they have private storages other apps can not access. However, IACs act as a bridge between apps, passing and receiving messages. Malicious apps exploit IAC mechanisms to trick components into performing unwanted actions, or hijack the message sent to a trusted component. Attack strategies for Android and iOS share a common idea, distinctive schemes have manifested as their design philosophies differ.

1) Intent Spoofing: Implicit intents can be received from any application that has a matching intent filter. In other words, activities that are started by implicit intents can always be started by any other apps as long as intent filters are defined accurately. The specifications of an implicit filter can easily be copied from Android manifest file in any Android APK. This is an extremely useful feature in Android, as developers can build more expandable apps. However, it can be abused to pose a threat to Android's security model [5]. Intent spoofing is an attack where an exported component receives an intent from unexpected source. As a real-world example, we analyzed the APK file of Amazon app and built an app that starts an activity for uploading data to Amazon

Cloud. If Amazon app does not authenticate the data, then its data storage in Cloud is open to a threat. That is, any activity that are started by implicit intents prone to injection attacks. In order to prevent this, a developer shouldn't assume that an activity always gets intents from benevolent senders, and they cannot rely on intent filters for securing components. It is recommended not to export a component if it is only for internal use. If an exported component performs operation on data that needs to be secured, a developer should authenticate a received intent to check if it has been sent from reliable application.

2) Intent Interception: The characteristic of implicit intents makes it particularly vulnerable for activity hijacking. Activity hijacking is done through a malicious activity designed to hijack an intent instead of an intended activity. For instance, if a mobile payment service starts a payment activity with implicit intent when a user, a malicious activity can intercept the intent and hijack sensitive data. IntentIntercept is an app that attempts to intercept as many implicit intents as possible. It has a long list of intent filters in its xml file and provide details of contents in an intercepted intent. With similar methods a malicious app can reveal sensitive data in intents by intercepting them between activities. However, intercepting an implicit intent is not always successful. When there are multiple applications that matches the implicit intent for an action, a user is prompted to select an application to start an activity, unless there is a default activity. Malicious apps usually disguise itself as a proper app to handle the action. Therefore, a user plays an important role in securing their own Android phone. Since many Android apps share implicit filters that are used to perform common tasks (i.e. creating an alarm, playing a media file, or opening a camera app), there are numerous implicit intents attackers can abuse to perform malevolent behaviors.

C. Security Threats to iOS

Even though iOS has restricted options and functionalities to communicate between apps compared to Android, security vulnerabilities still exist. Moreover, due to its closed nature, there are only a few public research for developers to refer to, especially regarding IACs. Attackers exploit poorly designed URL scheme logic to run an app without user's attention and to gain access to extract private information. Similar to Android, the registered URL scheme for any app is open to the public and can be acquired by opening plist file of an app package, which is iOS's equivalent of manifest file. An attacker can gain information on how an URL is handled, and come out with a malevolent string that will make an app to perform unwanted operations. If an attacker can lure a user to open a webpage with a script that calls an URL, the app will be launched from a web browser, sometimes even without asking the user's consent to do so. A popular example is the vulnerability of the older version of Skype [13]. When an URL `skype://0000000000?call` is called, Skype app starts a call without user's validation. The vulnerability still exists even if a user is prompted to choose whether they want to open an app or not. As the responsibility to filter malicious apps falls into users' hands, they have to be extra careful when accessing websites that can't be trusted.

VI. Threats to Validity

One threat to external validity is the generalizability of our experiment to other apps. Our apps test inter-app communication by sending information about a favorite color. Other apps may have different performance and security vulnerabilities based on how different they are from our app. We attempted to make this app as general as possible to avoid this threat to validity.

One threat to internal validity is the difference in size between the Android app and iOS app. The Android apps are 1.96 MB and 1.94 MB, while the iOS apps are only 330 KB and 250 KB. This is significant because larger apps may take longer to open, so it affects our comparison of IAC latencies between the Android and iOS implementations of our app. There also may be differences in the speed of the Android and iOS emulators used to test latency.

Regarding to the results of our experiments, the test results are generated and calculated by using `System.out.println()` in java and `print()` in Swift. Those two statements are both relatively slow I/O statement which may take time to print things on the console which might lead to different results. While this happens mainly due to the different operating system of Android and iOS. By trying to compare them with the performance, there is some mechanism behind them that can lead to a uncertainty results.

VII. Related Work

The analysis of Android and iOS inter-app communication each is a hot topic. But there is not really enough research on the compression and security analysis difference between Android and iOS. The Comparison of Inter-Application Communication Mechanisms in Mobile Operating Systems [2] have some part relating to the comparison of inter-app communication of different platforms. But what they focused on basically is only the simple usage and functionality difference between them. There is not really much about security and performance. In that paper, a comparison is carried out on the existing mechanisms for inter-application communication of different platforms including Android, iOS, Windows Phone and Blackberry. An existing limitation of current inter-app communication, which is the communication between cross-platform devices, was mentioned. They proposed an approach which is a framework for other systems to work together.

Another paper, Inter-app Communication in Android: Developer Challenges [1], talks more about how apps communicate with each other in Android platform. It introduced the Android platform-defined message mechanism. Specifically, they go through the third-party-contributed message types to mention about the obstacles for developers to attempt to use, which emphasize more on the aspect of developer

perspectives and developing process not the design and mechanism of Android itself.

As for security of Android inter-app application, Analyzing inter-application communication in Android [5] try to analyze the security and data violation issues that could happen due to the various inter-app message passing system. They examined Android application interaction and came up with a tool named ComDroid which is a tool to detect the vulnerabilities that an app can be attacked by an intent-base invasion.

VIII. Future Work

We have explored the performance and security aspects of the inter-app communication between Android and iOS. This has been the focus of the paper. A future deep dive into the security and performance differences between these two platforms may focus on the architecture and message delivery mechanisms. Since we have found a lot of mechanisms that are common, such as URI schema and security patterns, research on cross-platform application communication can build off of our findings. Additionally, we hope to continue our investigation by analyzing malware that makes use of security vulnerabilities in Android and iOS to gain more insight on securing IAC mechanisms.

IX. Conclusions

Android and iOS have different inter-app communication schemes. Android uses intents while iOS uses URL schemes to handle queries between apps. We have explored differences between Android and iOS in implementing inter-app communication by designing test apps to compare implicit intents, explicit intents, and URL schemes. We found that URL schemes have slightly lower latency compared to explicit intents, but iOS does not have an equivalent mechanism to implicit intents. Securing IAC mechanisms remains one of the most important challenges in the realm of mobile security. Spoofing attacks can be carried out on both Android and iOS apps, although attackers have fewer options for iOS. Intercepting a message is non-trivial for iOS but can easily be done on Android because of its design. Future research may focus on differences between Android and iOS to design solutions to these security issues.

References

[1] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. Inter-app communication in android: Developer challenges. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 177–188. IEEE, 2016.

[2] Kalaiselvi Arunachalam and Gopinath Ganapathy. The comparison of inter-application communication mechanisms in mobile operating systems. 2015.

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 63–68. ACM, 2011.

[4] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software*, 119:31–44, 2016.

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[6] Android Developers. Intents and intent filters, 2018.

[7] Android Developers. Sharing files, 2018.

[8] Android Documentation. Fileprovider, 2018.

[9] App Programming Guide for iOS. Inter-app communication, 2017.

[10] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security Conference*, pages 513–527. Springer, 2015.

[11] Charlie Miller. Mobile attacks and defense. *IEEE Security & Privacy*, 9(4):68–70, 2011.

[12] Ibtisam Mohamed and Dhiren Patel. Android vs ios security: A comparative study. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 725–730. IEEE, 2015.

[13] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John Lui. Enpublic apps: Security threats using ios enterprise and developer certificates. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 463–474. ACM, 2015.