CRiOS: Toward Large-Scale iOS Application Analysis

Damilola Orikogbo
Boston University
Boston, USA
dao@bu.edu

Matthias Büchler
Boston University
Boston, USA
mbuchler@bu.edu

Manuel Egele
Boston University
Boston, USA
megele@bu.edu

ABSTRACT

Mobile applications – or apps – are one of the main reasons for the unprecedented success smart phones and tablets have experienced over the last decade. Apps are the main interfaces that users deal with when engaging in online banking, checking travel itineraries, or browsing their social network profiles while on the go. Previous research has studied various aspects of mobile application security including data leakage and privilege escalation through confused deputy attacks. However, the vast majority of mobile application research targets Google's Android platform. Few research papers analyze iOS applications and those that focus on the Apple environment perform their analysis on comparatively small datasets (i.e., thousands in iOS vs. hundreds of thousands in Android). As these smaller datasets call into question how representative the gained results are, we propose, implement, and evaluate CRiOS, a fully-automated system that allows us to amass comprehensive datasets of iOS applications which we subject to *large-scale* analysis. To advance academic research into the iOS platform and its apps, we plan on releasing CRiOS as an open source project.

We also use CRiOS to aggregate a dataset of 43,404 iOS applications. Equipped with this dataset we analyze the collected apps to identify third-party libraries that are common among many applications. We also investigate the network communication endpoints referenced by the applications with respect to the endpoints' correct use of TLS/SSL certificates. In summary, we find that the average iOS application consists of 60.2% library classes and only 39.8% developer-authored content. Furthermore, we find that 9.32% of referenced network connection endpoints either entirely omit to cryptographically protect network communications or present untrustworthy SSL certificates.

CCS Concepts

•**Security and privacy** → **Software security engineering;** *Software reverse engineering;*

SPSM'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2994459.2994473

Keywords

ACM proceedings; iOS, Security

1. INTRODUCTION

The App store model is probably the most significant reason for the unprecedented success of mobile smart devices, such as those powered by Apple's iOS or Google's Android platforms. App stores allow third-party app developers to offer their software to customers of the respective ecosystem. As Apple and Google have a vested interest in protecting their customers (i.e., end-users) from malicious or misbehaving applications they have implemented various security mechanisms. For example, every application available on Apple's App Store goes through a vetting process that is designed to identify misbehaving applications. While both Apple and Google claim significant shares of the overall smart device market, existing security research predominantly focuses on the Android platform. While we are unaware of an authoritative reason that explains this imbalance, anecdotal evidence suggests that the comparatively open nature of the Android platform could have contributed to the current state of affairs. Unfortunately, this imbalance directly impacts the insights gained by the security community with respect to the two platforms. For example, while existing Android app analysis research is frequently evaluated with hundreds of thousands of applications (e.g., [19, 20, 25]), iOS security publications are commonly evaluated on much smaller datasets ranging from a few([15]), to hundreds([18]), to a few thousand([14, 17, 23]).

To start narrowing this gap of insights, this work focuses on the *large-scale analysis* of iOS applications. To this end, we identify the lack of a comprehensive app dataset as one of the most stringent limitations for iOS app analysis. This is intuitive considering the relative ease with which one can download Android applications, as opposed to digital rights management (DRM) protected delivery of iOS apps. Thus, previous work analyzing iOS apps relied on ad-hoc methods to collect applications from the official App Store. Furthermore, the mechanisms or datasets obtained by these works are not publicly available to others which is not conducive to advancing security research on the iOS platform.

To address this imbalance in datasets between Android and iOS we propose and implement a fully automated system that allows us to download an arbitrary large number of iOS applications from the Apple App Store. We use this capability to aggregate a dataset of 43,404 iOS applications. On the dataset collected, we perform various large-scale analysis such as library identification and an analysis of the

SSL-certificates used by the apps' communication endpoints.

In summary, this paper makes the following contributions:

- We present CRiOS, the implementation of our fully automated iOS application crawler. This system identifies iOS apps, downloads them and performs the necessary post-processing (e.g., reverting DRM protections) to convert the application into a representation suitable for further analysis.
- We evaluate CRiOS and aggregate a dataset of 43,404 iOS applications. This reflects 4.5% of the U.S. App Store. The size of the dataset and the fact that it spans all categories featured on the App Store are strong indicators that our findings are representative.
- As a first step toward the large-scale analysis of iOS applications, we perform library analysis on our dataset. This required us to address and solve challenges that are unique to the iOS ecosystem. Solving the library identification problem is necessary to correctly assess the impact of vulnerabilities and avoid over-counting in most bug finding analysis.
- As many applications interact with remote network endpoints, we use our dataset to identify these endpoints. We then analyze the security characteristics of these endpoints by analyzing the SSL/TLS certificates they serve.

2. APPROACH

At a high level we enable the large-scale analysis of iOS applications by providing two components. First, we present CRiOS, our system to collect a comprehensive dataset of iOS applications in Section 2.1. As the second step, Sections 2.2 and 2.3 then discuss the analysis we perform on the collected dataset.

2.1 Application Collecting Infrastructure

The first step in our large-scale analysis of iOS applications is the creation of a corresponding dataset. To aggregate this dataset we implemented an automated system comprising web crawlers and a set of iOS devices. More precisely, this automated system consists of three distinct parts:

1. Identify all applications available on the App Store
2. Download applications sequentially
3. Decrypt each application to enable further analysis

Figure 1 illustrates how the individual components of this system interact. First, the CRiOS controller builds an index of all applications available on the iTunes App Store. Subsequently, the controller sequentially instructs an instance of the iTunes client software to download applications from the App Store. The iTunes client is configured to store the downloaded applications as an `ipa`[1] file on shared storage. Each downloaded `ipa` file contains the application's binary executable and all resources needed for execution. Unfortunately, the binary executable is encrypted and thus cannot be used for static analysis. Thus, to decrypt the executable,

[1] `ipa` files are structured ZIP archives and are used to transfer applications from the App Store to an iDevice.

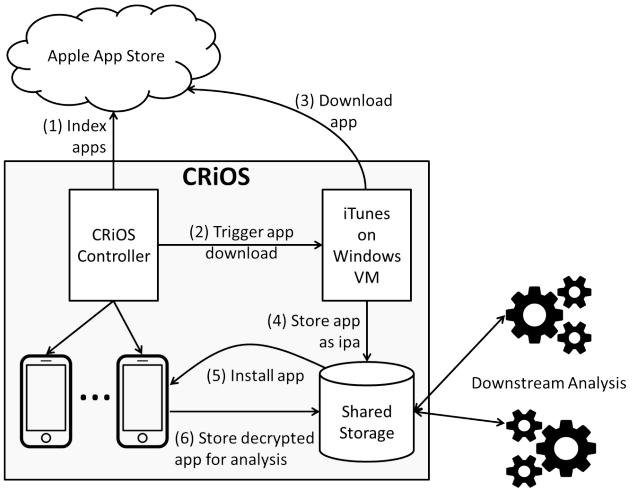

Figure 1: CRiOS system design for app collection

the CRiOS controller first installs the applications one-by-one on physical iOS devices. Although the file's content only ever exists on the file system in its encrypted state, the iOS loader will decrypt the binary (into memory only) during application launch. Similar to related work (e.g., [14, 18, 17]), CRiOS leverages this insight to attach to a running application and dump the decrypted contents to the file system. In the final step, CRiOS then stores the decrypted binaries on the shared storage. The result of this procedure is a set of applications which are suitable for the static analysis of iOS applications including those proposed in this paper.

2.2 Library Identification on iOS

An application consists of a set of classes it operates on. These classes come from different sources — they either originate from frameworks, external third-party libraries, or are developer-authored. Framework classes are not part of the application binary itself. Rather the corresponding framework is pre-installed on the iOS device. Only Apple, by means of an iOS update, has the capability of changing which frameworks are installed on a device. In contrast, both third-party library classes and developer-written classes are available in the binary itself. In order to prevent code duplication and profit from already implemented algorithms, apps usually make extensive use of external libraries for parts of their functionality. While only Apple can distribute and install frameworks, there are no such restrictions on regular third-party libraries. While Apple-provided libraries can be more or less conveniently collected[2], collecting third-party iOS libraries is much more difficult. Other approaches like Centroid [13] and AdRisk [22] make use of pre-defined lists of libraries. Being dependent on such lists is non-desirable because the complete list of available third-party libraries is not known in advance. Due to the lack of a convenient and centralized way of finding such third-party libraries, we propose an approach towards identifying such libraries using huge data sets of applications instead of pre-compiled lists. For our approach we perform the following steps to identify libraries used by iOS applications:

1. After the `ipa` file is downloaded from the Apple's app

[2] E.g., https://developer.apple.com/library/ios/navigation

store, we use a tool called `class-dump` [3] to extract a header file for every available class in the binary file. Such a file contains information about the class such as the class name, inheritance information, defined methods and categories. Having all class information available for every app provides us a 1:N relation between the app and the classes it uses.

2. We then track for every class a list of applications that contain the given class. Intuitively, developer-written classes only appear in 1 application whereas classes belonging to a library are likely to appear in different applications. Therefore the longer the list of applications for a given class, the bigger the likelihood that this class is part of a library.

3. To identify different libraries used by a given application, we collect all classes of that application that are used by at least two different applications. These classes are then clustered based on the prefix and the cohesion between classes. Initially, we put each class in its own cluster by building singleton clusters. Later, they are potentially merged in two different phases.

4. In phase 1, CRiOS merges two clusters in two cases:

   (a) All classes in both clusters share a common prefix of at least a predefined length and the prefix only consists of upper case letters. We leverage the fact that third-party library classes very often start with a prefix. E.g., classes belonging to the Facebook SDK for iOS very often begin with `FB`, classes belonging to the Google Mobile Ads SDK begin with `GAD`, PDTSimpleCalendar [12] names its classes with the prefix `PDT`, or the GPUImage library [9] names its classes with the prefix `GPU`.

   (b) All classes in both clusters share a common prefix of at least a predefined length (longer than in case 1) but may consist of lowercase letters as well. Intuitively, this case is motivated e.g., by the Flurry library [6] which names its classes with the prefix `Flurry*`. At the same time, classes like "OpenUDID" and "OperationManager" should not be merged based on the prefix only.

5. In phase 2, CRiOS considers the cohesion between any class c_1 of a cluster lc_1 and any class c_2 of a cluster lc_2. CRiOS merges clusters lc_1 and lc_2 if the average of the n largest cohesions is bigger than a predefined threshold value. For this step, CRiOS introduces a penalty factor for the cohesion depending on the length of the common prefix between two classes. Intuition dictates that the cohesion between classes that also share a common prefix has a higher weight than the cohesion between classes without a common prefix.

Finally, having a set of library classes for a given application allows us to analyze, how many classes of an application originate from a third-party library in our data set of 43,404 applications. If we further consider method invocations, the level of granularity can be increased to methods as well.

2.3 Network Communications

Mobile applications frequently communicate with back-end services over the network. The protocol of choice to implement these communications remains the HTTP protocol with its "secure" HTTPS counterpart. Although network communications of applications are interesting in and by themselves, this analysis is particularly timely considering Apple's announcement that they will ban applications if they use unencrypted (i.e.,HTTP) communications starting by fall 2016 [1]. Thus our results for apps that communicate via HTTP provide an insight into the developer efforts required to update existing applications to conform with these new requirements.

For our analysis of network connection endpoints, we perform the following steps:

1. We use regular expressions to extract all HTTPS and HTTP URLs in our dataset. This provides us a lower bound of resources an application references via secure (HTTPS) or insecure (HTTP) means.

2. We extract the domain names for all URLs in this set and in the case of HTTPS URLs, we query the corresponding hosts for any SSL certificates.

3. We analyze the host and certificate information obtained by the previous step to identify potential security issues.

As Apple will force developers in the future to abandon HTTP communications, we simply analyze how many applications rely on resources referenced via HTTP.

2.3.1 Analyzing Certificates

To establish a secure HTTPS connection, the server sends a certificate to the client to authenticate itself to the client. However, the connection can only be considered secure if the client verifies the authenticity of the certificate. For certificates signed by a trusted certification authority (CA), the iOS networking libraries will perform the necessary verification. While we optimistically assume that this verification logic is implemented correctly [8], we are mostly interested in certificates that are not signed by trusted CAs. It is common practice for development teams to develop against a back-end server that is only equipped with a self-signed certificate or signed by a CA that is not globally trusted. In these cases, the developer has to implement the verification logic themselves. An interesting study on Android found that these verification routines are "The most dangerous code in the world" [21]. As there is no reason to believe that iOS developers are better equipped to make security decisions than Android developers, we analyze the SSL certificates served by the network connection endpoints referenced by the applications in our dataset. To this end, we evaluate the following characteristics for each application and certificate we could retrieve:

- If the certificate is signed by a globally trusted CA, we assume the reference to the corresponding resource is secure.
- If the certificate cannot be verified by a trusted CA, we conjecture that the use might be insecure.
- Methods, such as certificate pinning allow developers to use self-signed certificates securely and thus the above characterization might be overly aggressive. Thus, we analyze the corresponding applications for known-insecure coding patterns that undermine the security offered by HTTPS connections.

3. IMPLEMENTATION

In this section we provide details about our implementation of the iOS app collecting system and the implemented evaluation strategies. In Section 4 we discuss experiments, results and implications based on these implementations.

3.1 The CRiOS System

As discussed in Section 2.1, the first component in CRiOS collects a large number of iOS applications in a manner suitable for static analysis. We discuss the implementation of the three necessary steps, (1.) identify, (2.) download and (3.) decrypt, in the following paragraphs. The full implementation of CRiOS will be made available under an open source license.

Identify Applications.

The primary interface to the App Store is the iTunes client software. Unfortunately, iTunes does not provide a list of all applications available in the store and only lists up to 200 apps for any of the 24 different categories. Apple's public figures indicate that the App Store contains over one million applications and thus the iTunes based approach is unfortunately insufficient. While the iTunes software remains the only way to install an application on an iOS device, Apple maintains web sites for all applications in the App Store. Thus, we created a simple crawler that enumerates these web resources and aggregates a list of all applications available on the App Store.

Download Applications.

Once we identify all applications on the App Store, we can iterate over all apps and download them locally. To the best of our knowledge, the only software that can download applications from Apple's App Store is iTunes. Thus, we configured a Windows virtual machine where we installed iTunes and logged into the App Store with a valid Apple ID. Furthermore, we equipped the virtual machine with the necessary functionality to automatically download an application via iTunes provided the application's identifier. This was simplified by the observation that iTunes registers a protocol handler for the `itms` URL scheme with the operating system. Any URL that references a location via the `itms` scheme will trigger iTunes to open at the corresponding location. As applications can be addressed as `itms` URLs, CRiOS can navigate iTunes to the corresponding app by opening the respective URL. For example, itms://itunes.apple.com/WebObjects/MZStore.woa/wa/viewSoftware?id=1094591345 will navigate to the Pokemon Go app in iTunes.

Once CRiOS navigates iTunes to the corresponding app page, CRiOS relies on the AutoIT [2] suite of GUI automation tools to click the install button. Note that clicking the button via proper Windows APIs (e.g., `SendMessage(hButton, BM_CLICK, ...)`) is unfortunately not an option. The reason is that the iTunes content pane is a webview rendered with webkit and thus the button is not a full-fledged GUI element. This additional layer of abstraction prevents the Windows APIs from directly accessing or clicking the button.

Once the AutoIT component clicks the button, iTunes will commence the download and store the application in the `ipa` format in the user's iTunes folder. The `ipa` format is a regular zip archive that bundles the application's executable with all required resources and meta-information.

Decrypting Application.

As mentioned previously, the executable contents of iOS applications are encrypted. Thus, before CRiOS can perform any meaningful analysis, it has to decrypt these contents. However, the only way we are aware of to decrypt an application requires that application to be installed on a real iOS device and then dump the memory contents of the running application. To integrate this functionality into the fully automated workflow implemented by CRiOS, we leveraged the libimobiledevice [10] library and the Clutch iOS application decryption utility [4]. More precisely, we leverage the capability of libimobiledevice to install applications in the `ipa` format on a USB-connected iOS device without further user intervention. Once the application is installed, Clutch is used to dump the decrypted executable contents of the application. Clutch achieves this functionality by first launching the application on the device. Upon program launch, the iOS kernel decrypts the application's content into memory. Clutch then attaches to the running process and dumps the now decrypted content to disk.

CRiOS implements this sequence of steps to automatically build a repository of iOS applications that can be subjected to further static analysis.

The Apple App Store and iDevices implement a stringent digital rights management (DRM) regiment. This DRM scheme ensures that Apple has full control over the iOS ecosystem including the devices, market place and applications. To implement these restrictions, Apple leveraged various cryptographic techniques and algorithms. More precisely, Apple uses cryptographic signatures to ensure integrity of the applications distributed through its App Store. Furthermore, the binary contents of each application are symmetrically encrypted. While the cryptographic signatures can be seen as a security mechanism for the user, the encryption of the executable serves no security purpose other than obfuscation. Despite no apparent security benefit, the encryption stops any attempts of statically analyzing iOS applications in its tracks.

3.2 Library Identification

A prominent feature of the mobile application environment is the prolific use of third-party libraries. The use of libraries itself is not unique to the mobile environment, that is to say commodity applications and systems also provide and use a plethora of libraries. However, what is characteristic for the mobile environment is the almost exclusive use of *statically* linked libraries while commodity applications predominantly link against dynamic libraries at runtime. The reason for this stark difference between mobile and commodity systems is that Apple does not allow applications to dynamically link against any libraries, other than the system-provided libraries (i.e., frameworks). Thus, two distribution models for third-party libraries exist for the iOS platform. That is, libraries can be distributed in source, or as static libraries in binary form. Developers then either add the library's source to their project, or instruct the linker to link the static binary blob into the resulting application. Irrespective of what approach the developer follows, the resulting binary mixes code that is authored by the developer and by the library author at the same time. However, this intermingling of provenance poses a challenge for the attribution of any findings an analysis system produces. For example, we need to ask for each detected vulnerability,

whether it is the result of a third party developer or whether the vulnerability affects thousands of applications because it is contained in a library. Furthermore, if we simply sum up the occurrences of vulnerabilities, results are prone to overcounting if they originate from frequently used libraries. To accurately reflect the impact of results from any analysis performed on mobile applications, we first need to identify which parts of an application's code are developer-authored and which parts are present due to third party libraries.

To identify libraries we follow a two step approach. First, we identify classes that appear in multiple applications. The rationale behind this step is the assumption that a library should occur in at least two applications. In the second step, we calculate how closely the classes from the first step interact with each other. The rationale for this step is that classes from any given library will interact more frequently than classes from different libraries. In fact, we would expect that classes from different libraries do not interact with each other as it is the developer's discretion that determines which libraries are linked with the application.

Same Class Multiple Apps.

To identify which classes occur in multiple applications we extract the *class signatures* of all classes in our data set. We define the class signature of a class analogously to the type signature for individual methods. A method's type signature contains the method's name, number, order, types of all arguments and return values. We define a class signature to contain the method signatures for all methods which are implemented by the class, as well as the names and types of all fields which are members of the class. CRiOS extracts the class signatures for all classes in an application with the help of the `class-dump` [3] utility. `class-dump` parses an iOS application for the class, method and field meta information of all classes in the app. This information is sufficient to extract the class signature for all classes in an application. However, to improve performance when comparing signatures of classes we first topologically sort methods and fields by their name and hash the result. We can then compare the hashes of the class signatures to quickly assess which classes occur in multiple applications and ignore the rest for the library identification as these are, by definition, not libraries.

A logical next step would be to identify groups of classes that always occur together in an application and declare those groups as libraries. Unfortunately, this approach is not sufficient as individual groups can easily contain classes from more than one library. For example, applications could include a Google library for advertising and the Flurry library for analytics. The above approach would incorrectly consider the union of classes of these libraries as a single entity. Also, if we consider two versions of the same library, the above scheme alone will fail due to the fact that the common classes between the versions will be considered a library and the parts that are unique to either version will be considered a library as well.

Ideally, however, we would like to identify individual versions of libraries in their entirety. The reason for identifying different versions of the same library is different versions can result in different findings in an analysis. For example, the library authors might fix bugs and vulnerabilities in newer versions of their code.

Class Cohesion.

In the above step we identified classes that are contained in multiple applications. However, to identify libraries themselves, we have to address the correct combination of different libraries and their versions. To this end, calculate a *class cohesion* metric between any of the identified groups of classes. We define class cohesion between two classes as the number of method calls between methods of these two classes. More formally, we denote the methods of class C_1 as M_1 and the methods of C_2 as M_2. We then define a binary relation *calls* as $(\rightsquigarrow) \mapsto M_1 \times M_2$ over M_1 and M_2 that indicates whether a method from M_1 can invoke a method from M_2. Finally, class cohesion (CC) is the number of calls between any methods in M_1 and M_2 respectively. That is,

$$CC(C_1, C_2) = |m_1^i \rightsquigarrow m_2^j| + |m_2^j \rightsquigarrow m_1^i| : m_1^i \in M_1, m_2^j \in M_2$$

Practically speaking we can easily identify method calls in an iOS application. However, identifying the callee (i.e., the target of a call or the m_2^j above) is a challenge. Solving this challenge would amount to re-constructing the applications' super control flow graph.

Instead of precisely solving for the *calls* relation we calculate a conservative over-approximation. To this end, we first extract the class hierarchy for all classes contained in an application. The class hierarchy captures inheritance relationships between classes, as well as the method signatures for all methods implemented in a class.

The message passing system in Objective-C requires that the selectors are passed as an argument to the dynamic dispatch routines in the runtime. A selector is the name used to select a method to be executed by an object and is a unique identifier that replaces the developer given method name when the source code is compiled. To pass a selector as an argument, it must be referenced prior to the call. Thus, our analysis first builds a relation that identifies for all methods the set of selectors used within the body of that method. In a second step, our analysis matches the inferred selectors with the information in the class hierarchy. More precisely, it identifies all methods whose name corresponds to the selector. This procedure is similar to the class hierarchy analysis proposed in [16]. With the *calls* relation evaluated for all methods in an application, we could simply calculate $CC(C_1, C_2)$ for all pairs of classes per application. However, our analysis is only concerned with library identification, thus we only need to compute CC for pairs of classes identified in the first step of the analysis.

Finally, we identify libraries as sets of classes whose class signatures occur in multiple applications and the class cohesion between pairs of such exceeds a threshold τ.

3.3 Network Communication

As mentioned in Section 2.3, we identify and analyze network resources that applications explicitly reference.

Extracting URLs.

To identify which network resources an application references we relied on regular expressions that match HTTP and HTTPS URLs. Due to its heuristic nature, this approach is prone to false positives and false negatives. As the regular expression matches strings in the application in a context-insensitive manner, references to an identified URL could be confined to dead code in the program. As dead code will never execute, the URL would never be requested

at runtime. False negatives arise when an application uses, for example, string concatenation to construct a URL at runtime, instead of statically storing the URL in the binary. However, false negatives of this kind only arise if the protocol scheme and the domain name are combined at runtime. In most cases we observed, the application stores the protocol and domain name as a prefix and then creates the remaining parts (e.g., path, query parameters and anchor) of the URL dynamically. In all these cases, the regular expression will identify the correct value.

After identifying the URLs contained in our dataset, we grouped them with respect to scheme and domain name.

Using the results from the previous step, we determined the fraction of apps that connected to each external resource. As 100% of the applications refer to Apple's servers to obtain certificates used to verify signed application content, we removed references to any Apple domains from further processing.

HTTPS and HTTP Accesses.

Based on the above results we classified each application into one of three groups:

- Application exclusively references HTTPS resources
- Application exclusively references HTTP resources
- Application uses both protocols

3.3.1 HTTPS Certificates

The use of HTTPS provides the facilities to establish secure connections. However, this requires that the server sends a certificate to the client for authentication. We assume an application correctly verifies a server's certificate if the certificate is signed by a trusted CA. The reason for this trust is based on the observation that for such certificates, the validation is performed by the iOS network libraries and the developer does not have to interfere. Thus, for all extracted HTTPS URLs we perform the following steps.

1. Extract and visit the domain from the URL
2. Obtain server-provided certificate
3. Attempt to validate the certificate

To fetch the certificate chains from domains referenced via HTTPS URLs we used OpenSSL's `s_client` SSL/TLS client program [11]. At the same time, we attempted to verify the certificate chains and recorded any errors if the certificate chains could not be fully validated.

4. EVALUATION

In this section we first provide details about the collected iOS applications. Subsequently, we discuss the results we obtained by applying our analysis to the dataset.

4.1 Application Dataset

We continuously ran CRiOS for a period of roughly 4 weeks on the U.S. App Store. During this time, we identified a total of 957,237 applications and downloaded 88,480 or 9.2%. CRiOS attempted to decrypt the executable content of all these apps, and succeeded for 45,482 to provide us with a dataset that spans 4.8% of the U.S. App Store. Unfortunately, CRiOS was not able to decrypt all of the applications we downloaded. This is due to various reasons, such as device incompatibility, or iOS version requirements our setup does not fulfill. More specifically, our current setup of CRiOS is equipped with two iPod Touch devices. Thus, applications that require iPhone or iPad devices cannot be installed or analyzed on our infrastructure. However, we can easily alleviate this restriction by adding devices from these categories to our setup. Furthermore, there is no iPod application out of the 88,480 apps that does not run on an iPhone. Furthermore, the latest iOS version installed on the devices in our current environment is iOS 8.1.1. Thus, applications that require newer versions of iOS cannot be analyzed with our current setup either. Recall that CRiOS requires that the iOS devices it uses are jailbroken. We can address the iOS version limitation similarly to the device limitation stated above. Provided a jailbreak for newer iOS versions, we can simply add devices that run these newer versions of iOS to CRiOS without further changes. Table 1 details these numbers broken down by categories, how many applications we could download, how many we analyzed, and how many applications can be obtained free of charge.

4.2 HTTPS vs. HTTP

Of the 43,404 applications we analyzed, we found that 96% referenced a total of 69,504 unique domains. As shown in Table 2, 14,265 of the 69,504 references were to HTTPS URLs, while 55,239 were to URLs via the HTTP scheme. Upon further inspection of the 41,713 applications that referenced external resources, 31,055 applications made connections to secure HTTPS services while 40,866 applications made connections to insecure HTTP domains.

Table 3 shows the classification of the applications and their use of the different transfer protocols. 25.6% of the applications reference external resources strictly via HTTP, while only 2.80% rely strictly on HTTPS.

4.3 SSL Certificates

As part of the analysis, CRiOS attempts to determine how many HTTPS servers present valid certificates issued by well-known CAs. Beyond certificates that can be easily verified, we wanted to determine the prevalence of self-signed certificates among the network connection endpoints referenced by the apps in our dataset. As described in Section 4, we requested the certificates from all domains referenced via HTTPS URLs. While validating the resulting certificates, we encountered eight different error codes. Figure 2 shows the distribution of domains that returned the different errors and Table 4 explains the cause of the errors.

Of the 14,265 HTTPS servers requested, we were able to connect to 80% (11,996) of them and examine the certificates. A closer inspection revealed that about 2% (218) of the domains had expired certificates (one in particular dating back to 2006) and 1.75% (210) of the domains had a self-signed certificate anywhere in the certificate chain. Figure 3 shows the range of expiration dates on the certificates from the HTTPS servers. From the range of expiration dates, we can see that some developers are accessing servers with expired certificates and are not validating the HTTPS certificates properly.

For applications within our dataset that referenced HTTPS resources (31,055) shown in Table 3, we queried the appli-

Category	on iTunes	Free (%)	Downloaded (%)	Analyzed (%)
Games	262,176	187,353 (71.5)	18,964 (7.2)	8,691 (3.3)
Entertainment	99,576	72,523 (72.8)	13,066 (13.1)	6,705 (6.7)
Education	95,945	60,614 (63.2)	8,372 (8.7)	3,942 (4.1)
Lifestyle	81,899	67,906 (82.9)	8,434 (10.3)	4,326 (5.3)
Productivity	52,667	39,129 (74.3)	5,059 (9.6)	2,678 (5.1)
Business	33,295	32,299 (97.0)	2,200 (6.6)	1,180 (3.5)
Books	32,259	15,815 (49.0)	3,290 (10.2)	1,428 (4.4)
Finance	30,405	26,423 (86.9)	3,406 (11.2)	2,024 (6.7)
Health & Fitness	29,187	20,994 (71.9)	2,007 (6.9)	1,100 (3.8)
Food & Drink	28,568	24,335 (85.2)	1,555 (5.4)	907 (3.2)
Medical	28,200	20,170 (71.5)	2,588 (9.2)	1,380 (4.9)
Utilities	26,557	21,813 (82.1)	2,954 (11.1)	1,591 (6.0)
Navigation	23,030	12,371 (53.7)	2,482 (10.8)	1,650 (7.2)
Travel	18,468	15,045 (81.5)	1,744 (9.4)	1,010 (5.5)
News	18,429	17,014 (92.3)	3,312 (18.0)	1,980 (10.7)
Reference	18,358	13,307 (72.5)	2,419 (13.2)	1,143 (6.2)
Photo & Video	17,714	11,541 (65.2)	1,235 (7.0)	624 (3.5)
Social Networking	12,166	10,932 (89.9)	1,168 (9.6)	645 (5.3)
Music	11,692	8,476 (72.5)	1,315 (11.2)	743 (6.4)
Magazines & Newspapers	11,386	11,185 (98.2)	1,219 (10.7)	731 (6.4)
Sports	10,986	8,438 (76.8)	934 (8.5)	536 (4.9)
Catalogs	10,445	9,374 (89.7)	338 (3.2)	207 (2.0)
Shopping	3,296	3,218 (97.6)	354 (10.7)	217 (6.6)
Weather	533	373 (70.0)	65 (12.2)	44 (8.3)
Total	957,237	710,648 (74.2)	88,480 (9.2)	45,482 (4.8)

Table 1: Breakdown of applications by category on iTunes and in our dataset

Unique Domains	
HTTP	55,239
HTTPS	14,265
Total	69,504

Table 2: Number of unique domains referenced by HTTP and HTTPS URLs

Protocol Scheme	# of Applications
HTTP	10,658
HTTPS	847
Both	30,208
Total	41,713

Table 3: HTTP(S) resources referenced in apps

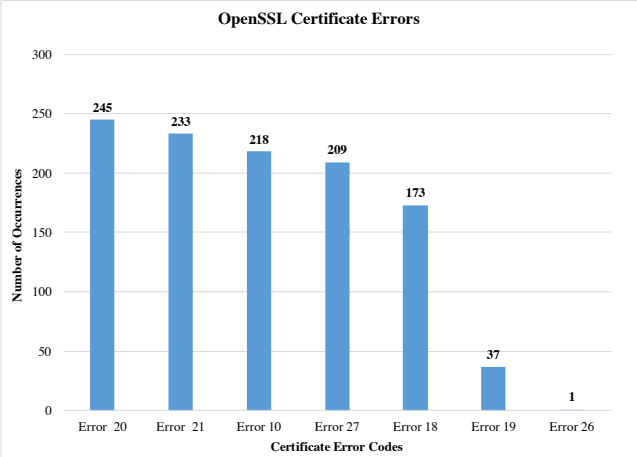

Figure 2: Number of domains that output SSL errors during certificate validation

Error Code	Description
0	Verification OK
10	Certificate has expired
18	Self signed certificate
19	Self signed certificate in certificate chain
20	Unable to get local issuer certificate
21	Unable to verify the first certificate
26	Unsupported certificate purpose
27	Certificate not trusted

Table 4: Description of OpenSSL error codes

cation for known insecure certificate validation method calls to see whether the certificate validation is actively bypassed by the developer. More specifically, we analyzed whether the corresponding applications invoke `allowsAnyHTTPSCertificateForHost` or `setAllowsAnyHTTPSCertificate`. We examined our dataset and discovered that 1% (344) applications made use of the iOS methods.

4.4 Third-party Libraries

As described in Section 1 we initially downloaded 43,404 applications from the app store. Out of these applications, 43,059 application binaries can be successfully processed by `class-dump` [3] and therefore can be used for identifying libraries. For this evaluation, we considered four different libraries: GPUImage [9], PDTSimpleCalendar [12], Flurry Analytics [6] and Google Analytics [7]. The GPUImage library [9] is open source. We downloaded two different versions of this library. *Core-v1* refers to the GIT commit ID 167b038 on May 26, 2016, whereas *Core-v2* refers to the GIT commit ID 56300d1 on Feb 23, 2015. Open source libraries very often contain example applications and test cases as well. It is likely that a developer removes such additional resources before releasing the app, so we evaluate our approach on the core library if possible. The GPUImage core

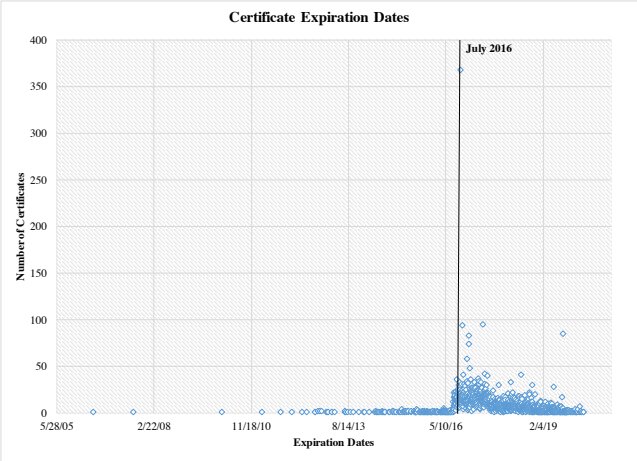

Figure 3: Aggregate expiration dates of SSL certificates

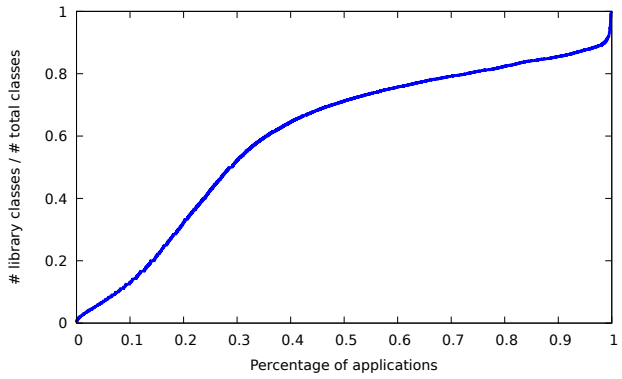

Figure 4: CDF of Library Classes over Applications

library (Core-v1) contains 173 classes, while Core-v2 contains 165 classes. The PDTSimpleCalendar library [12] is open source as well. The core library with the GIT commit ID d79e4ce contains 5 classes. In contrast, Flurry [6] and Google Analytics [7] are closed source libraries. To evaluate our approach on closed code libraries, we need a list of classnames that belong to these libraries. Since `class-dump` does not support analyzing static libraries directly, we compiled two standard sample applications which include Flurry and Google Analytics respectively. In the next step, we used `class-dump` to analyze these sample applications. Using this approach, Flurry (version 7.6.6) consists of 73 classes, whereas Google Analytics (version 3.16.0) consists of 250 classes.

As described in Section 2.2, CRiOS identifies libraries by generating sets of classes (called library footprints), that represent these libraries. To evaluate the quality of these library footprints, we randomly selected for each library an iOS application from which we know by manual inspection, that the application makes use of the given library. CRiOS then generates the library footprints for these applications and we compare them to the list of library classes.

For each application, we generate four different sets. TP is the set intersection between the classes determined by CRiOS and the downloaded library. FP is the set difference between the classes determined by CRiOS and the downloaded library. These classes are reported by CRiOS but are not part of the downloaded library. FN is the set difference between classes belonging to the downloaded library and classes reported by CRiOS. These classes are missed by CRiOS, although they belong to the library. Finally, TN is the set of classes of the application that belong to neither the library nor are reported by CRiOS. With these sets, we calculate the True Positive Rate as $TPR = \frac{|TP|}{|TP|+|FN|}$ and the False Positive Rate as $FPR = \frac{|FP|}{|FP|+|TN|}$. Table 5 lists the selected applications and the corresponding results.

For app2, CRiOS generates a library footprint that is identical to the set of classes of the PDTSimpleCalendar library. For app1, the TPR is 97.2% with a FPR of 0.0%. Manually inspecting the false negative classes, CRiOS misses two classes with the prefix `PodsDummy`. These classes are assumed to be library classes, because our approach to generate a list of classes from closed code libraries uses cocoapods [5], while we assume that app1 does not. For app3, CRiOS reports a TPR of 96.96% because the false negative set contains 5 classes all starting with the prefix `GPUImage*` that are all missed. Therefore, app3 most likely use a different version of the GPUImage library than we considered for this evaluation because `class-dump` does not report them in the first step. Nevertheless, if `class-dump` had reported them, CRiOS would have included them in the footprint, since they share an uppercase prefix of length 4. For app4, CRiOS reports a TPR of 91.9% with a FPR of 0.46%. App4 contains a class called `GPUimageDirectionalSobelEdgeDetectionFilter`, which is reported as a false positive. This classname contains a typo (char 4 is a lowercase i instead of an uppercase i) and is corrected in *Core-v1*, but not in *Core-v2*. Since Objective-C inherits the case sensitivity from C, CRiOS considers these as different classes. Therefore, this class is not reported as a false positive using *Core-v2*. Finally, CRiOS reports a TPR of 97.6% and a FPR of 0.34% for app5. This app is particularly interesting because it demonstrates the necessity of combining the two strategies based on prefixes and cohesion. For the Google Analytics library, CRiOS combines several classes with no shared prefix at all to the same library footprint (e.g., `GSD*`, `FIR*`, `ACP*` and `GAI*`). For this app, CRiOS reports a false positive because it includes a class named `FIRInstanceID-FIRApp` in the library footprint, which is not part of the downloaded version of Google Analytics. Based on the classname, it is very likely this class is part of the Google Analytics library, but probably in a different version. In terms of false negatives, CRiOS reports 6 classes (`FIRAAudienceComparisonValues`, `FIRANetworkLogger`, `GGLConfiguration`, `GMRConfiguration`, `PodsDummy_Pods_test`, `GGLContext`). Three classes (`FIR*`) are not reported by `class-dump` in the first step. The other three classes are missed in the phase where the cohesion is used to merge different sets of classes.

Finally, we calculated the fraction of library classes out of all identified classes for each of the 43,059 applications. Figure 4 shows the cumulative distribution function of library classes over applications. The figure illustrates that about half of the applications consist of up to 71.2% library classes, while the average of library classes over all 43,059 iOS applications is 60.2%. Furthermore, 70.9% of all apps consist of more library classes than developer-written classes. Our

	AppName	Version	ID	Category	Library	TPR	FPR
1	Fertilizer Removal by Crop	1.4.2	520209986	Reference	Flurry	0.9726	0.0
2	WVCCU Mobile Banking	3.0.0	500495547	Finance	PDTSimpleCalendar	1.0	0.0
3	Live Kaleidoscope Free	2.1	364924172	Photo & Video	GPUImage (v2)	0.9696	0.0
4	Live Kaleidoscope Free	2.1	364924172	Photo & Video	GPUImage (v1)	0.919	0.0046
5	Radio1 - La première FM de Tahiti	3.1	368606346	Music	Google Analytics	0.976	0.0034

Table 5: iOS Applications for Library Evaluation

analysis further shows that about 569 applications consist of more than 90% library classes and 107 applications are reported to consist of library classes only. An explanation for this number could be that all developer-authored code is provided in the skeleton class(es) that XCode generates or the application logic is written in C or C++.

5. DISCUSSION

As our CRiOS operates on real-world large-scale iOS apps, it is subject to internal and external threats to validity.

Internal Threats to Validity. Of particular concern regarding threats to the internal validity of our analysis is the *calls* relation we use to determine the class cohesion metric described in Section 3.2. Our approximation for the *calls* relation is based exclusively on the method's name and a name collision could lead to the incorrect grouping of an unrelated class in a library cluster. For this to occur, the name collision must occur with the methods of a legitimate class in the cluster and thus, libraries are identified to contain spurious member classes.

External Threats to Validity. Our dataset and therefore analysis exclusively consists of applications that can be downloaded from the App Store for free. It is thus possible that the same analysis we conducted on these free apps could result in different results if the dataset consisted of paid applications. For example, developers of paid applications might be more diligent when encrypting network communications or paid applications might contain different amounts of developer-authored or library code. While the constitution of the dataset can have such effects, the methods we presented should be equally applicable to paid applications too.

6. RELATED WORK

Viennot et al. [25] performed an analysis of Android applications available in the Google Play store. They developed a crawler that downloaded over 1.1M applications. The analysis mainly focuses on the evolution of these apps over time, the library usage and its impact on application portability, duplications of apps and the ineffectiveness of authentication mechanisms. While Viennot et al. [25] focus on Android Java applications, our work considers iOS applications. Thus, our approach needs to decrypt the `.text` segment of an iOS application binary with the user's private key. In terms of security tokens, we do not rely on known formats of such tokens but tried to identify tokens where the syntax is not known a priori. Furthermore, we analyzed the applications with respect to HTTP vs. HTTPS connections and inspected the corresponding certificates for potential security breaches.

Egele et al. [18] analyzed iOS applications specifically for privacy leaks. As part of their contribution, they developed a novel approach and a tool called PiOS. Their approach is based on a static data flow analysis and the authors applied it on a data set of 1,400 iPhone applications. Egele et al. [18]'s approach differs in regard to CRiOS in that we contribute with a scalable crawler for iOS applications that allowed us to download and decrypt more than 40k applications. Therefore the applied evaluation metrics are based on a much bigger set of apps and allows us to consider multiple security aspects.

Ma et al. [24] published a third-party library detection approach for Android applications. LibRadar uses static analysis techniques to detect such libraries. Both LibRadar and our approach consider binaries and do not consider precompiled lists of known third-party libraries. Furthermore, both approaches are clustering- and hashing-based. Nevertheless, identifying libraries fundamentally differs for Android and iOS applications because iOS applications do not know the concept of package hierarchies and Objective-C applications are message-based. The latter is crucial for precisely determining cohesion between classes. Since our approach for iOS applications can not leverage hierarchies, we consider class cohesion information. Finally, while Ma et al. [24] use hashing of so called static code features, characteristics that cannot be obfuscated, our approach uses hashing not on the basis of class interactions but to identify unique class signatures, including method, property and interface signatures.

Chen et al. [14] published an approach for identifying potentially harmful libraries both on Android and iOS. It is based on the fact that many libraries are offered for both systems. The approach first builds clusters of similar packages in Android applications since library identification is very simple for Android applications. These libraries are then analyzed with the help of anti-virus services to identify harmful ones. By considering invariant features between Android and iOS libraries, the corresponding iOS version of the harmful Android library is identified. To conclude that the iOS library is harmful, the approach maps suspicious behavior in the iOS library to corresponding behavior in the Android variant. Our work substantially differs by the fact that our approach identifies libraries directly from the iOS binary, without the detour to corresponding Android libraries.

Chen et al. [13] also published an approach to detect clones in Android applications. The approach determines the similarity between methods leveraging geometry characteristics of dependency graphs. Both our and Chen et al. [13]'s approach share the task to identify third-party libraries, but differ because Chen et al. [13] address Android applications, and uses library whitelists. Our approach does not depend on such pre-defined lists.

Finally, Grace et al. [22] published an approach which requires the identification of third-party libraries as well. For this approach, our work differs almost in the same way as for

Chen et al. [13]. Grace et al. [22] approach is again based on a pre-defined list of 100 distinct ad libraries. Furthermore, the approach is dedicated to Android applications.

7. CONCLUSION

Large-scale mobile application analysis is mainly confined to the Android ecosystem. We observe that one of the reasons might be that collecting a representative dataset of iOS applications is significantly more challenging to achieve. To narrow this gap, we present CRiOS, a system which lets us aggregate large datasets of iOS applications suitable for subsequent analysis. With the help of CRiOS we collect a dataset of 43,404 iOS applications and subject them to two large-scale analysis. First, we perform library identification, and find that an average iOS application consists of 60.2% library classes. We also analyze the network connection endpoints that applications reference to identify which applications need to be updated to conform to Apple's upcoming HTTPS-only requirements. Furthermore, we analyze how these endpoints use SSL certificates to secure communications with their mobile applications, and detect apps that might validate SSL certificates incorrectly. By making our implementation of CRiOS available to the community, we hope to positively affect the research into iOS applications.

References

[1] https://techcrunch.com/2016/06/14/apple-will-require-https-connections-for-ios-apps-by-the-end-of-2016.

[2] https://www.autoitscript.com/site/autoit.

[3] http://stevenygard.com/projects/class-dump.

[4] https://github.com/KJCracks/Clutch.

[5] https://cocoapods.org.

[6] https://developer.yahoo.com/flurry/docs/analytics.

[7] https://www.google.com/analytics.

[8] https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch.

[9] https://github.com/BradLarson/GPUImage.

[10] http://www.libimobiledevice.org.

[11] https://www.openssl.org/docs/manmaster/apps/s_client.html.

[12] https://github.com/jivesoftware/PDTSimpleCalendar.

[13] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE. ACM, 2014.

[14] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *37th IEEE Symposium on Security and Privacy*, IEEE S&P, 2016.

[15] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-r. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *19th Annual Network and Distributed System Security Symposium*, NDSS. The Internet Society, 2012.

[16] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP. Springer-Verlag, 1995.

[17] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iRiS: Vetting Private API Abuse in iOS Applications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS. ACM, 2015.

[18] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS. The Internet Society, 2011.

[19] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS. ACM, 2013.

[20] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. CLAPP: Characterizing Loops in Android Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE. ACM, 2015.

[21] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS. ACM, 2012.

[22] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC. ACM, 2012.

[23] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *20th Annual Network and Distributed System Security Symposium*, NDSS. The Internet Society, 2013.

[24] Z. Ma, H. Wang, Y. Guo, and X. Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE. ACM, 2016.

[25] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS. ACM, 2014.