Studying TLS Usage in Android Apps

Abbas Razaghpanah
Stony Brook University
arazaghpanah@cs.stonybrook.edu

Arian Akhavan Niaki
Stony Brook University
sakhavanniak@cs.stonybrook.edu

Narseo Vallina-Rodriguez
IMDEA Networks / ICSI
narseo.vallina@imdea.org

Srikanth Sundaresan
Princeton University
srikanths@princeton.edu

Johanna Amann
ICSI
johanna@icir.org

Phillipa Gill
U. Massachusetts – Amherst
phillipa@cs.umass.edu

ABSTRACT

Transport Layer Security (TLS), has become the *de-facto* standard for secure Internet communication. When used correctly, it provides secure data transfer, but used incorrectly, it can leave users vulnerable to attacks while giving them a false sense of security. Numerous efforts have studied the adoption of TLS (and its predecessor, SSL) and its use in the desktop ecosystem, attacks, and vulnerabilities in both desktop clients and servers. However, there is a dearth of knowledge of how TLS is used in mobile platforms. In this paper we use data collected by Lumen, a mobile measurement platform, to analyze how 7,258 Android apps use TLS in the wild. We analyze and fingerprint handshake messages to characterize the TLS APIs and libraries that apps use, and also evaluate weaknesses. We see that about 84% of apps use default OS APIs for TLS. Many apps use third-party TLS libraries; in some cases they are forced to do so because of restricted Android capabilities. Our analysis shows that both approaches have limitations, and that improving TLS security in mobile is not straightforward. Apps that use their own TLS configurations may have vulnerabilities due to developer inexperience, but apps that use OS defaults are vulnerable to certain attacks if the OS is out of date, even if the apps themselves are up to date. We also study certificate verification, and see low prevalence of security measures such as certificate pinning, even among high-risk apps such as those providing financial services, though we did observe major third-party tracking and advertisement services deploying certificate pinning.

CCS CONCEPTS

• **Security and privacy → Security protocols**;

KEYWORDS

Security Protocols, Android, Mobile, Network Measurements, Transport Layer Security, Secure Sockets Layer, SSL, TLS, Transport Layer Protocols, Mobile Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '17, December 12–15, 2017, Incheon, Republic of Korea

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5422-6/17/12…$15.00
https://doi.org/10.1145/3143361.3143400

ACM Reference Format:
Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. 2017. Studying TLS Usage in Android Apps. In *Proceedings of CoNEXT '17*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3143361.3143400

1 INTRODUCTION

With users entrusting increasingly sensitive data to their mobile devices, there has been a push for deployment of secure communication protocols. Transport Layer Security (TLS) and its predecessor, Secure Socket Layer (SSL) are the most widely used protocols to encrypt network communication, with Google observing an increase from 50% to 85% of connections to their servers using TLS between January 2014 and April 2017 [45] and Mozilla reporting that more than half of observed connections use TLS [97]. Apple requires iOS applications to use TLS [60].

However, deploying TLS is not a magic bullet: its end-to-end architecture exposes many options that could render a bad implementation insecure, *e.g.*, weak ciphers, or compromised Certificate Authorities in the Public Key Infrastructure (PKI). In the desktop era, the surface area of attacks on TLS had been mostly restricted to a handful of browsers developed by large, well-resourced organizations. However, the "app-store" model of modern mobile OSes vastly increases risk: we now have millions of devices that may not have up-to-date OSes or apps; even those that do may run apps developed by hundreds of thousands of developers who may not understand the complexities of TLS. It is therefore critical to understand how TLS is being used by mobile apps.

Previous efforts to understand the TLS and PKI ecosystem have mostly relied on (i) active network scans which gives a large-scale and longitudinal view of TLS at the server side but not the client side, thus missing the huge variety of client devices, apps, and versions [27, 30, 34, 84, 84, 91], (ii) static analysis of applications' source-code which may not cover all possible code paths [41], (iii) or dynamic analysis which requires manually inspecting apps –hence limiting scale– in a dedicated network testbed using an in-path MITM proxy [38].

In this paper, we present the first (to our knowledge) holistic and large-scale study of how Android apps use TLS in the wild. We obtain rich, but anonymized, data from Lumen Privacy Monitor, a free Android app that we developed which monitors network traffic locally on the device in user space, without requiring modifications to the mobile OS. Lumen has the ability to collect data from normal user-app interactions, map network flows to apps, and collect rich TLS handshake data. Lumen is available to download for free from the Play Store and allows users to monitor their apps'

online activities and privacy leakages. Our dataset spans more than 5,000 Lumen users who downloaded the app from the Play Store and 1,364,420 TLS handshakes generated by 7,258 apps, connecting to 34,176 servers.

Using this data, we study:

(i) *The impact of using different libraries and configurations:* We fingerprint OS versions, TLS libraries, and apps using cipher suite lists, and show that a vast majority of apps (84% of app-versions in our data) use OS default TLS libraries. 2.3% of apps use OpenSSL, and another 13% use other alternative libraries or change default settings of known libraries. These implementation choices have security implications: apps that depend on OS default libraries and configurations, are vulnerable if the OS itself is insecure (*e.g.*, Android 4.1.1 is vulnerable to Heartbleed). According to Google, over 60% of Android users run outdated OS versions [46]. On the other hand, apps that use their own TLS libraries are vulnerable to poor implementations due to developer inexperience: we see 32 app-versions announcing null ciphers (no encryption), and 32 app-versions announcing anonymous ciphers (vulnerable to downgrade and man-in-the-middle attacks).

(ii) *Certificate verification:* We dig into the mechanics and usage of X.509 certificate validation across apps. We analyze how apps react to the presence of Lumen's in-path man-in-the-middle (MITM) TLS proxy [78] to identify those vulnerable or robust to TLS interception. Specifically, we look at the use of bundled CA stores, self-signed certificates, and certificate pinning. Our findings show that these practices are not common in mobile apps; less than 2% of all apps perform either certificate pinning or CA certificate bundling in order to prevent TLS interception.

(iii) *Vulnerable TLS implementations:* Using previous findings about TLS vulnerabilities, we identify apps that are potentially vulnerable to attacks, based on traces of their TLS handshakes. Although most apps running on newer versions of Android likely have fewer vulnerabilities, misconfigurations by developers can still leave applications vulnerable. Our findings show that apps created by well-resourced companies with dedicated security teams tend to use better and more secure TLS configurations.

Our focus is to study the client-side of TLS usage in Android, as a comprehensive server-side analysis would be orthogonal to the goal of this paper and would require active measurements on the servers. Moreover, while it is possible to perform a similar study in iOS (and possibly other mobile platforms), we do not have a tool similar to Lumen in iOS or any other mobile OS, and thus our study is limited to Android. Our results paint a complex picture of TLS on Android and suggest that the path towards better security is murky at best. We discuss how the TLS ecosystem could be improved by making updates to Android's TLS APIs more reliable, providing better access to developers using Android's internal TLS APIs to configure parameters, making sure OS-default trust stores remain uncontaminated and reliable, and finally, making sure app developers are properly trained and aware of security guidelines and best-practices.

In § 3 and § 4 we introduce Lumen Privacy Monitor and the dataset. In § 5 we analyze client-side TLS handshakes and extract default TLS library fingerprints for various versions of Android OS and third-party TLS libraries like OpenSSL and GnuTLS, and use them to characterize how mobile app developers and third-party libraries (*e.g.*, ad and analytics services) implement TLS. We also study how apps use TLS extensions. In § 6, we study vulnerabilities and weaknesses in different versions of TLS libraries used by apps due to misconfigurations and developer errors. In § 7 we analyze X.509 certificates and study how apps implement techniques such as certificate pinning and CA bundling to detect and prevent TLS interception proxies. We conclude our study with a discussion on the current use of TLS by mobile applications and its implications on mobile traffic security (§ 9).

Finally, in the interest of data transparency and hoping that the research community will benefit from this data in furthering the research work in this area, we have made our dataset available to the public. [1]

2 BACKGROUND: ANDROID AND TLS

Transport Layer Security (TLS) is the current IETF standard that provides a secure and reliable encrypted connection between two end-points over an untrusted network. To make TLS deployment simpler, Android provides application developers with the option of using a native TLS library that is shipped with the Android OS. Apps also have the option of using third-party or their own libraries.

A History of TLS Support in Android: Android has supported TLS 1.0 since its first version released in 2008 and TLS 1.1 and TLS 1.2 since 2012. However, it does not provide native support for TLS 1.3 as of writing (Android 7.1). Android's TLS support has also evolved over OS releases, adding support for different TLS extensions across versions [2] and removing old cipher suites, extensions and parameters which had become obsolete [19]. OpenSSL was Android's default TLS provider [21] until it was replaced by BoringSSL, an OpenSSL fork, in Android 6.0 [4, 17]. Consequently, the security guarantees of TLS library features vary across OS versions. While these changes should ideally be widely applied in the form of OS updates and security patches, in reality, Thomas *et al.* [86] have shown that many Android devices do not receive security updates in time, while many others never receive any at all. This causes mobile apps using default TLS APIs using vulnerable cipher suites and SSL/TLS versions on older devices vulnerable, regardless of how up-to-date the app itself might be.

How Apps Use TLS in Android: Application developers can either opt to use Android's native TLS libraries or use third-party TLS libraries ("providers" [7, 35]). GnuTLS [42] and OpenSSL are two such providers. Android also allows app developers to implement and use their own proprietary TLS libraries: *e.g.*, Firefox's Network Security Services (NSS) [71]. Third-party providers are limited to the current app process, and therefore their use does not affect the system or other applications [35].

Outdated Default Libraries: According to Google's own data on Android version usage, as of May 2017, 61.7% of Android phones are running Android versions 5.x (released in late 2014) and lower, which are considered to be outdated versions of Android with equally outdated TLS APIs. Many such phones will no longer receive security updates, as Google itself does not guarantee security

[1] The dataset and instructions on how to use the data are available at `https://haystack.mobi`.

updates to its own Nexus 4 phone, whose last supported version is Android 5.1.1, after November 2015 [46]. As a result, apps using default OS-provided TLS libraries can unwittingly use old and weak ciphers suites and protocol versions, and even be subject to well-known vulnerabilities when they run on outdated devices, regardless of how new the app itself might be. For instance, apps running on devices still using unpatched Android 4.1.1 are vulnerable to Heartbleed [74].

This state of affairs may influence security conscious application developers to bundle a pre-built TLS library with their application package (*e.g.*, OpenSSL, NSS, GnuTLS) to gain control and ensure a more consistent TLS configuration across different devices and OS versions.

Root Stores: As of Android 7, Android's official Android Open Source Project (AOSP) source code has a list of over a 150 trusted CAs that are audited and updated in each release. Android also provides support for third-party root certificate installation either manually from the system settings or programmatically [5]. This feature is often required by enterprise networks or VPN-based services to perform legitimate TLS interception [57, 78]. In the rare case that a CA is breached and issues certificates for a hostname to an adversary, Android contains a built-in blacklist in the operating system for these certificates and CAs. The root store can vary across different Android OS vendors, some of them adding over 50 additional certificates to the trusted root store [89]. Since Android version 7, any app that targets API level 24 (*i.e.*, Android 7.0) and above does not trust any user-added CAs for TLS connections by default (developers must explicitly allow it), thus impeding TLS interception [18].

Securing TLS Traffic: Using TLS sockets does not necessarily guarantee a secure connection over the Internet. Mobile app developers need to ensure that their TLS connections are properly configured: poor configurations allowing weak cipher suites and vulnerable TLS versions are detrimental to the security of a TLS connection. Most TLS libraries, including Android's native one, allow developers to specify and configure certain parameters of a TLS connection and adapt them to the requirements of the application. These range from supported protocol versions and cipher suites to TLS extensions such as server name indication (SNI).

Once the TLS handshake is completed and the client has received the X.509 certificate from the server, the developer is responsible for verifying the authenticity and validity of the server certificate using techniques like certificate pinning [7]. In fact, Android allows app developers to programmatically trust a specific set of CAs, rather than the operating system one by bundling a custom trusted cert store with their app [6]. Unfortunately, many developers may not correctly use Android's TLS libraries, either due to intended decisions (*e.g.*, improve app's performance due to the time required to validate the sessions [72]) or due to lack of technical background, thus allowing malicious in-path entities to intercept app communications using TLS interception techniques [16, 28, 66, 75].

3 LUMEN OVERVIEW

The Lumen Privacy Monitor app, available to download for free via Google Play [65], is a tool that helps users to understand how mobile apps handle their private data. In this section, we overview the design and operation of Lumen and how we leverage its ability to collect anonymized handshake data to analyze how Android apps use TLS in the wild.

3.1 Capturing Traffic Locally in User-Space

Lumen acts as middleware between apps and the network interface: it leverages the Android VPN permission and implements a simplified network stack to capture and analyze mobile traffic locally on the device and in user-space, without requiring root permissions. By operating in user space, Lumen can be distributed through traditional app markets and installed by any user, as long as the device is supported. [2] This enables large-scale crowd-sourcing, with the concomitant benefits of enabling traffic measurements with real user stimuli (as opposed to virtual environments [3, 98]), and also increasing our app coverage by several orders of magnitude compared to previous research efforts [38].

By running locally on the device, Lumen is also able to correlate disparate and rich contextual information, such as app identifiers, system version and process IDs, with flows: *e.g.*, it can match outgoing flows with the process that generated them, and with other device metadata such as the device vendor, model, Android OS version, and even TLS proxy exceptions. The access to metadata is a key feature in Lumen that gives us the ability to attribute our observations to apps, third-party libraries (*e.g.*, ad networks and analytics services) and even to device vendors. Note that Lumen does not forward any payload to a remote server for analysis, hence minimizing the privacy and security risks for the user; and keeping the network path unmodified.

We point readers interested in Lumen's architecture to our previous work [78]. We have made steady improvements to Lumen: the current version (v2.0, released in October 2017) can process traffic at over 95 Mbps rate when connected over a WiFi link, with a battery overhead below 1%.

3.2 The Lumen TLS Proxy

Lumen implements a *local* TLS interception proxy to analyze encrypted flows on the device, with user consent, to detect and report personal information leakages of apps to the users. Lumen's local MITM TLS proxy requests the user to add a self-signed root certificate (generated at install time) in the system trust store. Lumen parses network flows and identifies TLS/SSL handshakes, which it then forwards —along with flow-level meta-information the proxy requires in order to connect to the server (*e.g.*, IP address, port, SNI)—to the TLS proxy to set up the proxy connection. While we are not using the TLS interception facilities of Lumen (*e.g.*, privacy leakage reporting) for this work, we do collect and analyze stack traces and error information generated by proxy connection failures to measure apps' resistance to MITM attacks in presence of rogue CA certificates in the trust store. Note that, throughout this paper, the term "app" refers to a unique app package name.

TLS Proxy Impact on Apps: We try to minimize the side-effects of our proxy on mobile applications. Since the proxy relies on

[2] *e.g.*, Samsung's KNOX API, present in some Samsung devices, interferes with the normal behavior of Android's VPN interface.

Android's native TLS library, which does not support certain parameters, cipher suites, or TLS extensions, we cannot transparently proxy all flows. Applications that rely on certificate properties, extensions, or cipher suites not supported by base Android may cause proxied connections to fail. Note that we intentionally upgrade vulnerable SSLv3 [62] connections to the remote server when connecting to hosts that support newer protocols.

This limits Lumen's ability to be fully transparent both to the client app and the remote server. Lumen whitelists apps that it cannot proxy so that subsequent connections will be directly forwarded to the destination server. Nevertheless, those failed connections provide valuable data to identify and study apps implementing certificate/public-key pinning or CA-certificate bundling, which are not visible to a passive observer in the network (§ 7).

Collecting Handshake Data: Due to this lack of complete transparency by our TLS proxy, we limit the data collected from proxied TLS flows to the original "Client Hello" (CH) received from the app, and the server certificate chain, both of which are unaffected by the proxy. Furthermore, clients running Lumen version 1.3 (released April 2017) periodically skip proxying connections to get a full picture of the untouched TLS handshake without presence of a proxy, and deployment of security measures such as certificate pinning.

The collected anonymized handshake messages are sent to a backend server, which parses and extracts relevant information from the handshake messages using a purpose-built parser that uses a comprehensive list of cipher suites, extensions, and other parameters collected from IANA's list of TLS parameters [55], RFC drafts, and other sources like the source code of popular implementations of TLS and SSL like OpenSSL and GnuTLS. This information allows us to identify parameters that have been made obsolete, have had the code representing them in TLS messages repurposed for newer parameters, or are in active use despite not being standardized yet. In total, we obtained a map of 349 cipher suites, 34 extensions, 12 application layer protocol negotiation values (ALPNs), 37 elliptic curves, and 30 signature algorithms and the corresponding codes representing them in handshake records from these sources.

3.3 Ethical Considerations

Examining user traffic raises ethical issues that we consider carefully. We ensure data collected for this study only includes anonymized TLS/SSL handshake messages that are not encrypted and therefore do not reveal the contents of the user's Internet traffic. Lumen preserves user anonymity by not uploading any major browser activity (which we can test without user stimuli), traffic payload, or any user or device identifiers to our servers. In § 4 we describe the data collected by Lumen. Our tool only performs TLS interception with user consent after thoroughly informing the user about its purpose and implications. We provide user-friendly explanations of the purpose of the tool and the data collection process on the Google Play listing, through the privacy policy, our project website, and on the app. Additional precautions and considerations are also detailed in our previous work[78]. Our institutional IRB considers our effort as non-human subjects research—*i.e.*, we analyze the behavior of software, not people.

4 DATA OVERVIEW

Between November 2015 and June 2017, more than 5,000 users from over 100 countries have installed Lumen Privacy Monitor directly from Google Play. This has allowed us to analyze 1,486,082 TLS connections from 7,258 apps connecting to 34,176 domains (identified by their SNI) across 250 TCP ports (§ 4.2). We were able to collect TLS traffic traces for more than 891 unique tuples of Android SDK version number, device vendor, and model. [3] Further, we collected 684,209 TLS proxy exceptions associated with 4,268 apps and 10,753 domains.

Mobile apps are regularly updated with new versions to fix bugs, improve usability, or add new features [20]. It is therefore critical to study app behavior across several versions. Crowd-sourcing mobile traffic data allows us to collect traffic for several versions of a given app. Although our data is sparse in terms of app coverage due to its user-dependency, we could successfully obtain various app versions for 28% of the apps and at least 10 versions for 2.7% of the apps.

4.1 App Representativity

To understand the nature of the observed applications, we download meta-data about each app in our dataset from the Google Play Store. 3% of the apps in our dataset are paid apps, 85% are free and 12% are not listed in the Google Play Store. The set of apps not available on Google Play range from basic Android services to apps obtained from alternative app stores (*e.g.*, MoboGenie [69]), removed apps (*e.g.*, Free WhatsDog), and pre-installed apps and services from a number of mobile OS vendors (*e.g.*, LG, Samsung, and CyanogenMod among many others). Our monitored apps fall under 35 different Google Play categories. The two most commonly seen categories are: Games (16%) and Tools (9%). Excluding pre-installed services, 40% and 2.4% of the apps in our records have more than 1M and 100M installs, respectively. Our data covers 87 of the top 100 free apps on Google Play.

4.2 Traffic Statistics

88% of the mobile apps in our dataset communicate with online services over HTTPS. However, other protocols such as secure email (42 apps) and Google's Cloud Messaging service for push notifications (9 apps) [11, 47] also use TLS. Despite the fact that the increasing number of apps using TLS seems like a positive trend, 50% of the apps use simultaneously HTTP and HTTPS.

We have found 178 apps opening TLS sessions in non-conventional TLS ports, including 20 apps using TLS on TCP port 80. A close look at the distribution of TCP ports per app reveals traffic patterns that resembles of P2P traffic. We have identified this behavior in apps like Facebook, Skype, the official Tor client and apps possibly establishing P2P links using WebRTC APIs [94]. Furthermore, we identified 36 mobile apps interacting with other devices and services hosted on the local network (*i.e.*, they connect to IANA-reserved private IP addresses [61]). The majority of those apps use proprietary protocols such as Google's Chromecast clients (TCP:8008 and 8009), Smart-TV remote controllers (*e.g.*,

[3] We started collecting the device vendor and SDK version number with Lumen version 1.0.11 (Released in July 2016). We use the SDK version number as a proxy for the OS version running on the device. The ability to extract the exact OS version has been included in Lumen version 1.3.

Android's TV remote, TCP:6466, and LG TV Plus, TCP:3001), and apps to remotely access and configure the CPE (*e.g.*, MyFRITZ!App, TCP:49443).

5 CLIENT-SIDE HANDSHAKE ANALYSIS

We analyze CH messages for each flow and application and analyze the cipher suite list, the use of TLS extensions, and how third-party services (*e.g.*, ad networks) use TLS.

5.1 Method: Developing TLS Library Fingerprints

TLS libraries and OS APIs modify their supported cipher suites across versions: new cipher suites are added, weaker ones removed, and the order of their preference changed over time. As a result, both the set *and* order of the cipher suites found in CH messages is a strong signal to identify the TLS library–and its version–used by the application and reveal their vulnerabilities. Durumeric *et al.* used this concept to passively distinguish HTTPS requests generated from specific web browsers and versions [31].

We first build a corpus of cipher suite list fingerprints paired with their corresponding OSes and libraries. For that, we use a custom Android app that opens TLS connections using different TLS libraries—*i.e.*, Android's native library and two well-known third-party TLS libraries, OpenSSL, and GnuTLS—to a server under our control. We then extract the cipher suite list fingerprint from the CH message. We run this app on multiple Android phones running stock Android version 4.0 to version 7.1. We did not test older versions of Android as Lumen does not support these versions. We extract two cipher suite lists for each TLS library and OS version: the *default list*—*i.e.*, when a TLS connection is set up without explicitly configuring the cipher suite list—and the *supported list*—*i.e.*, the list of all cipher suites, including those not enabled by default, that are supported by the library on a given Android version.

Our analysis reveals that each TLS library and OS version has a unique cipher suite list. Additionally, we observe that some Android security patches modify the list of ciphers, but the OS reports the same OS version number. Figure 1 shows the cipher suite lists for three OpenSSL versions and three Android OS versions. The color code represents their order of preference; therefore if a suite is present in two versions but has different ordering, the fingerprint will still be different. As an example, the latest Android 7.x (SDK 24 and SDK 25) default cipher suite list has changed significantly from Android 4.x, removing 26 old and deprecated cipher suites (including all old SSLv3 and RC4 ciphers), while adding 10 new cipher suites, including the ChaCha20/Poly1305 family of mobile-friendly cipher suites which have been widely adopted by Google in recent years [48]. Minor revisions for the same library do not vary much, sometimes, just slightly reordering preferences based on shifts in adoption, or changes in codes representing cipher suites. For example, initial cipher suite lists for Android 6.x and 7.x devices showed announcement of temporarily-assigned codes for ChaCha20/Poly1305 ciphers, while later patches changed those to the permanent codes assigned after standardization of these ciphers. For validation purposes, we compare our harvested cipher suite fingerprints with those available in SSLLabs's collection of TLS user agent capabilities [77], finding identical fingerprints for those indexed on SSLLabs.

5.2 TLS Library Usage in the Wild

We use our fingerprint database to match the handshake sequences passively collected by Lumen. Our fingerprints allow us to easily distinguish apps that use OS APIs and OpenSSL with default settings. However, it is harder to distinguish apps that use other libraries or significantly change the cipher suite list by adding, removing, or changing the preference order of cipher suites; we label those apps under the "Other/Unclassified" label.

Figure 2 gives a breakdown of libraries and APIs in TLS flows. According to our results, 84% of app-version combinations use default settings on OS-provided APIs, and 2.3% of app-versions use various versions of OpenSSL. We could not match the remaining 13% app-versions to a known library fingerprint. We manually classified handshake traces for those cases in which we did not have OS default fingerprints available and study each app on a case-by-case basis. We provide salient findings and examples for apps falling in the "others/unclassified" category in the following paragraphs.

Apps Changing Default Settings: Apps that configure the connections with parameters other than the default ones include Uber, Facebook, and Microsoft apps (among others). While some of these apps alter parameters for better security (*e.g.*, Facebook removes RC4, 3DES, and other vulnerable cipher suites from OpenSSL list of cipher suites), others do it in a way that arguably makes TLS connections less secure. For example, all analyzed versions of BlackBerry Messenger [15], Viber [92], Wire [96], and Jio4GVoice [58] voice-over-IP (VoIP) use a short list of 1 to 3 cipher suites in TCP-based TLS connections to some of their servers, none of which provide forward-secret key exchange algorithms. Such ciphers allow a passive observer that records encrypted communications of these connections and has the ability to access the private keys of these servers to be able to decrypt all encrypted communications offline, provided that there are no other encryptions used on top of TLS. [4] While this might be impractical for most in-path attackers, it is a feasible attack vector for nation-state adversaries capable of both passively collecting large amounts of encrypted traffic and gaining access to private keys using legal means and court subpoenas.

One particularly interesting app in our dataset is the Hola VPN app [53]. Hola uses a P2P scheme to enable VPN-like functionality: any participating node can act as an egress point for any other participant. Since it forwards application layer traffic from other phones to the Internet, the app can have different TLS fingerprints on the same phone, representing different OS versions and libraries based on how many other phones are using it as a proxy for their traffic.

Third-party TLS Library Usage: Apart from OpenSSL, we could only identify two other third-party libraries that are used in our data: two apps use GnuTLS in at least one of their versions (VLC player [93] and SoundCloud) while Firefox and Firefox-derived

[4] This does not extend to UDP-based TLS connections of these apps, as we do not study UDP flows in this paper.

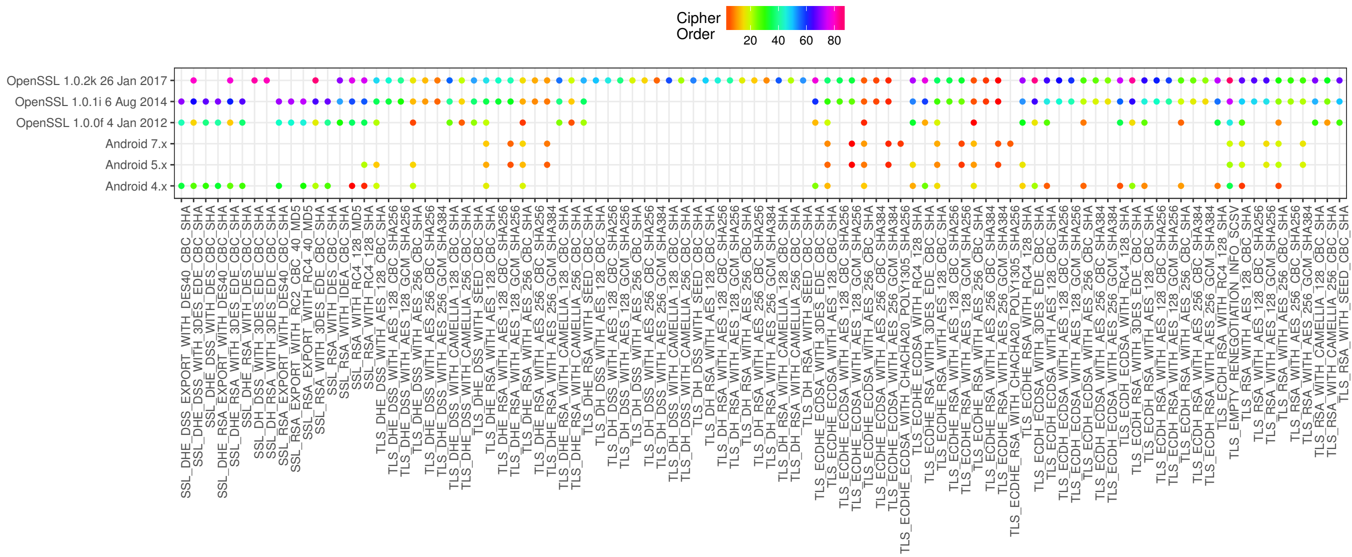

Figure 1: Comparison of the TLS cipher-suite lists for Android versions 4.X, 5.X, and 7.X and three different OpenSSL versions.

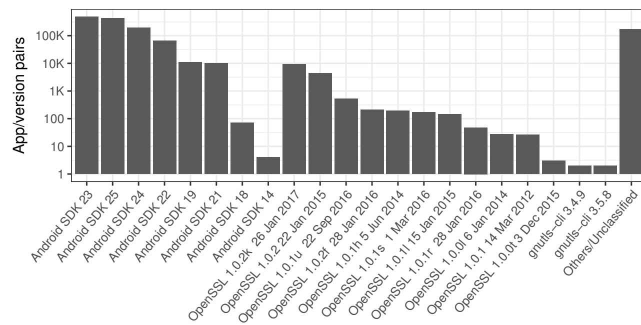

Figure 2: Rank of TLS libraries in TLS flows (log10-scale). Most flows are generated by OS-provided APIs with default settings.

browsers use NSS [71]. There are a number of other apps that either bundle other TLS libraries we cannot identify, or change the default parameters: Samsung Internet Browser [81], Chromium-based browsers, and Dolphin browser [26]. Furthermore, apps like Twitter, Dropbox, Spotify, and Uber have their own unique cipher suite lists in order to communicate with first-party domains. Some games and entertainment app makers (*e.g.*, EA [33], Miniclip [68], and Rovio [80] games) also have their own unique fingerprints, which could point to game engine code using other internal libraries for TLS. Microsoft apps (*e.g.*, Office apps) have their own unique fingerprints depending on the type of application (*e.g.*, cloud storage apps vs. Skype), which are in some cases close to OpenSSL default cipher suite lists.

5.3 The Case for Longitudinal Studies

Most research studies analyze a snapshot of the most popular apps at a given time. However, mobile apps may evolve and use different libraries and TLS configurations across app versions to gain control over their TLS traffic and security guarantees. Our data does not contain a large enough sample for most apps, with only 2.7% of the apps having 10 versions or more. This limits our ability to analyze changes in TLS usage behavior over time across all apps. However, even within the time-frame of our data, we have found examples of popular apps evolving their use of TLS. This demonstrates a need to study apps longitudinally to analyze and evaluate developer decisions over time and to gauge developers' awareness and experience with respect to TLS usage. While these popular apps use TLSv1.2 by default, they mainly differ in their lists of supported cipher suites and extensions. We now list two examples of such apps.

Twitter: Our data shows that the Twitter app matches OS-default SDK fingerprints prior to September of 2016. However, since September 2016, while fingerprints of connections it makes to third-party analytics services (*e.g.*, Crashlytics [22]) still match that of OS defaults, flows destined for Twitter's first-party domains have a different cipher suite list. This list differs from the OS-default one in two key ways: (1) it deprioritizes ChaCha20/Poly1305 cipher suites, that are given preference to all other cipher suites in the OS-default list due to their mobile-friendliness, behind the more traditional and more widely-supported AES-GCM/SHA family of cipher suites; and (2) it removed cipher suites that use non-elliptic-curve variants of the Ephemeral Diffie-Hellman (DHE) key exchange algorithm. An active TLS scan of `mobile.twitter.com` using SSLLab's scan tool [82] shows this decision to be in line with Twitter's server-side support for cipher suites, as it does not support ChaCha20/Poly1305 ciphers, or the non-EC variant of DHE.

Facebook: Facebook and Facebook Messenger have used OpenSSL to communicate with Facebook servers in all versions that are present in our data. However, Facebook's use of OpenSSL has evolved over time, with newer versions removing weak or unsupported cipher suites from their default list. Removed ciphers include Seed, DES, Export, and Camellia families of cipher suites, with the latest being DES ciphers in March 2016. Facebook also uses its own versions of HTTP/2 and SPDY in the list of application

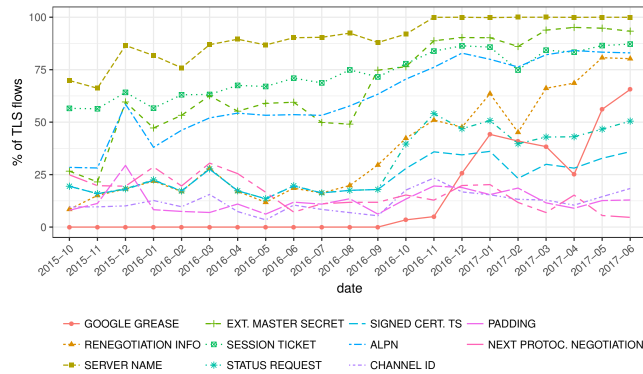

Figure 3: TLS extension usage over time, excluding 14 extensions that are near 100% or 0% across the board.

layer protocol negotiation protocols (ALPN). We discuss ALPNs in § 5.4.

5.4 TLS Extensions

TLS clients can use extensions to relay support for specific protocol features, like supported signature algorithms and supported elliptic curves, or to provide additional information like the requested hostname or the application layer protocols supported. This mechanism allows TLS to add new connection parameters without changing the base protocol. In total, we see 34 different extensions being used. Figure 3 shows use of 11 such extensions in TLS flows over time. We do not include ones that are present in more than 99% of the connections (*e.g.*, elliptic curves extensions and signature algorithms), as well as marginal ones that do not appear in more than 1% of the flows (*e.g.*, pre-shared key exchange modes extension and draft TLSv1.3 extensions). We describe notable cases below.

Google GREASE: Google started drafting Generate Random Extensions And Sustain Extensibility (GREASE) in 2016 [24], as a way of testing TLS servers for potential interoperability issues arising from Google's use of non-standard TLS parameters and extensions. To do this, they insert a set of non-standard and unassigned identification codes in various places (*e.g.*, cipher suites and extension list) in the CH to see whether the servers ignore them, or terminate the connection due to poor implementation. Any apps that use Android's default TLS libraries have GREASE parameters in their CH records. Our data shows GREASE usage in TLS extensions going from 0% of TLS flows in October 2016, to over 65% of the total TLS flows in June of 2017. We will revisit GREASE in § 9.

Application Layer Protocol Negotiation (ALPN): The ALPN extension (standardized in 2014) allows clients to negotiate the next protocol used on top of TLS (*e.g.*, HTTP/2 or HTTP/1.1). Using ALPNs in TLS handshakes allows the negotiation of the next protocol to piggy-back on the TLS handshake, saving a few round-trips after the TLS connection has been established, which in turn increases responsiveness in mobile apps and decreases load times of mobile web pages. Overall, our dataset shows wide adoption of this extension, from appearing in 28% of all flows in October of 2015, to 83% in June of 2017. This indicates that more mobile apps supporting multiple application layer protocols could now

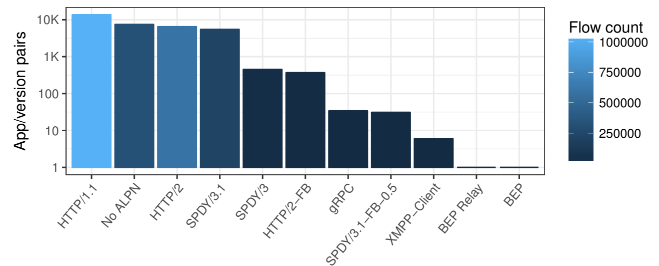

Figure 4: Application layer protocol support, negotiated via ALPN extension.

establish connections with servers faster. There are 15 ALPN values standardized by IANA (*e.g.*, HTTP/1.1, HTTP/2, and SPDY), and we have identified 10 ALPN values used by our corpus of mobile apps. Of those 10, 6 are not standardized yet by IANA with 420 app-versions using them.

Figure 4 shows the adoption of ALPNs across app-versions. Android's OS APIs enable HTTP/1.1 (and more recently, HTTP/2) by default for apps using internal APIs for HTTPS connections, which explains the high percentage of app-versions (75.99%) and flows (82.77%) supporting HTTP/1.1. Google apps (*e.g.*, Chrome and Google Maps) announce support for HTTP/2 and SPDY/3.1 as well as HTTP/1.1, with 39.27% of app-versions and 42.87% of flows announcing support for HTTP/2 and 33.26% of app-versions and 15.52% of flows supporting SPDY/3.1. Additionally, 34 app-versions from Google use `grpc-exp` [50], a Remote Procedure Call protocol, to communicate with its public announcement servers (*e.g.*, personal safety and emergency assistance, and phone spam protection) as well as Google's Federated Learning [67] machine learning servers (*e.g.*, Google Keyboard predictions and learning data collection).

Facebook apps use their own custom versions of HTTP/2 and SPDY/3.1, as denoted by `http2-fb` and `spdy/3.1-fb-0.5`, while Syncthing, a file synchronization app, uses an open-source protocol called Block Exchange Protocol (BEP [85]).

5.5 Third-party Services

Mobile apps may embed third-party libraries in their source-code to add features such as bug reporting, analytics, and support ads in their apps. Examples of these libraries are Crashlytics [22], Google Analytics [44], Flurry [40], and Appsflyer [10]. All major advertisement services and analytics APIs with the exception of Facebook Graph API [37] use OS-default APIs with default settings across the board. This is also true of apps like Twitter that use different TLS configurations for TLS connections that are made to destinations other than those related to third-party services. The most significant exceptions are the Facebook Graph API—which uses OpenSSL, similar to all Facebook-related apps, and the Sky API (related to SkyDrive). Finally, third-party libraries like Crashlytics allow for certificate pinning in their APIs, even when the app using these APIs does not necessarily pin its own certificates. We discuss certificate pinning in more depth in § 7.4.

To sum up, this analysis shows that app developers may not have much control on how third-party libraries connect to on-line

services over TLS, thus adding diversity and complexity to TLS usage within a single app.

6 TLS MISCONFIGURATIONS AND VULNERABILITIES

In this section, we study weaknesses in TLS implementation by looking at presence of weak cipher suites and vulnerable protocol versions in the TLS handshakes.

6.1 SSLv3 Usage

TLS clients can announce support for a range of protocol versions by setting the TLS record layer version to the lowest supported and the CH version to the highest supported version [25]. Android SDK versions 23 (Android 6.0) and newer do not announce support for SSLv3 by default, while older versions (Android 5.1 and older) do so. Android 5.1 introduced measures to prevent downgrade attacks, even if apps announce support for SSLv3 [8, 70]. This means that apps using default Android TLS libraries running on Android versions lower than 5.1 will be vulnerable to downgrade attacks. We also see 31 app-versions of 24 unique apps announcing support for SSLv3 despite running on devices that do not announce it by default, possibly due to developer error. Among them are the most recent versions of mobile games made by EA Games [33] (*e.g.*, FIFA Mobile, Madden NFL Mobile, *etc.*) with hundreds of millions in combined installs.

6.2 Weak Cipher Suites

It is well known that server-side support of weak and vulnerable ciphers is insecure and should be avoided [51, 63, 101]. Despite the fact that TLS/SSL protocols use transcript hashes to ensure the integrity of handshake, a recent attack, named SLOTH [13], makes it possible for a powerful adversary to downgrade a connection to any cipher suite advertised by the client. This means that announcing support for weaker cipher suites by clients can expose them to downgrade attacks. Although sometimes the mere presence of certain weak ciphers (*e.g.*, RC4) in the cipher list does not necessarily mean that the app is vulnerable, the best way of preventing attacks is by disabling weaker ciphers entirely.

Looking at server hello messages reveals that cipher suites chosen by the servers are strong across the board even when weaker ciphers are announced by the client. However, since an active scan of the servers at the time of each handshake is not available to us, we can not confirm whether servers also supported these vulnerable cipher suites. Below we list the most concerning cases of weak ciphers announced by apps.

Null Ciphers: Null ciphers provide no encryption and, when used, allow data to be sent in the clear, making them not secure by definition since an in-path adversary, under certain circumstances, can perform a down-grade attack to force a misconfigured client app and server to use them and then eavesdrop on their non-encrypted communications. They should always be disabled and never used in production settings. Despite this, 32 app-versions of 19 unique apps leave them enabled. Current versions of TuneIn Radio [88] and K-9 Mail [59], both with millions of installs, have these ciphers enabled. Another notable example is the security-focused VPN client F-Secure Freedome VPN [36].

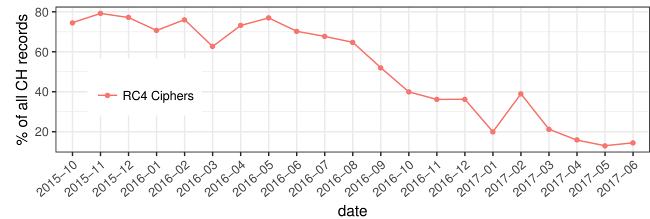

Figure 5: A temporal view of RC4 cipher usage in our data. Our data shows a decline in use of RC4 since February 2015 when RC4 was announced vulnerable.

Anonymous Ciphers: Some cipher suite lists contain anonymous ciphers that do not provide server authentication. When these cipher suites are used (*e.g.*, through a down-grade attack), they leave the client vulnerable to MITM attacks as the server side can be impersonated by an in-path adversary due to this lack of authentication. We find 32 app-versions of 19 unique apps that announce these ciphers along with other non-anonymous ciphers. Examples include most recent versions of Super Mario Run [83] (with tens of millions of installs), and, once again, TuneIn Radio.

Export-grade Ciphers: Export-grade, or "Export40" ciphers, are a family of old cipher suites that, due to old US regulations, use key lengths of 40-bits or shorter to be deemed legal for export to foreign countries (this is no longer the case). The short crypto key length in these cipher suites makes them unsuitable for use. Older versions of Android supported Export40 ciphers, which makes apps running on them vulnerable to the FREAK attack [49]. In total we see examples of 50 app-versions announcing these cipher suites on OSes that do not announce support for them by default. Apps like Tiffany Alvord Dream World [87], a children's game with in-app purchases that has over one million installs, uses a poorly configured OpenSSL library that enables a variety of weak ciphers, including Export40 family of ciphers.

RC4 Ciphers: Multiple attacks targeting RC4 were found in recent years, leading app vendors and TLS servers to drop support for it as per RFC-7465 [76]. Android and Google products have also dropped support for these ciphers, which caused a drop in the number of RC4 ciphers announced in TLS flows in our data (Figure 5). Yet, apps that use OS-default TLS settings still announce support for these cipher suites when running on Android versions 5.1 and older. Moreover, 872 app-versions of 311 unique apps announce support for them despite running on newer versions of the OS that have removed these cipher suites from their default cipher lists.

7 CERTIFICATE ANALYSIS

To better understand how certificates and public keys are utilized in servers used by mobile apps, we analyze server certificates and apps use of different certificate verification methods.

Root Certificate Authority	Count
Google Internet Authority G2	1990
Symantec Class 3 Secure Server CA - G4	1688
Go Daddy Secure Certificate Authority - G2	1675
COMODO RSA Domain Validation Secure Server CA	1549
DigiCert SHA2 High Assurance Server CA	1326

Table 1: Top 5 most popular root CAs in our dataset, sorted by the number of certificates signed by each.

7.1 Server Certificate Key Sizes

In order to have secure server side authentication, the server's public key has to be of a size that is not easily factorizable. RSA keys with 1024 bits of length or more satisfy this requirement, although NIST recommends use of RSA keys of at least 2048 bits of length [12], or EC keys that are at least 256 bits long. Note that an EC key length of 256 bits provides similar security as a 3072-bit RSA key. Our data shows that out of 21,189 total certificates, 99% have key sizes that provide security that is equivalent to or better than a 2048-bit RSA key (or 256-bit EC key), as recommended by NIST [12], while 0.7% have lower-size keys.

7.2 Root Certificate Authorities

Table 1 shows the top 5 CAs signing the largest number of certificates in our dataset. Google Internet Authority G2 signs the largest number of certificates, all of which belong to Google and its services and apps. This is to be expected due to the number of Google apps and services installed on Android devices. The rest of the top CAs differ from previous observations of TLS server certificates on the Internet as a whole [91]. This shows that some CAs are more relevant in the mobile ecosystem. For example, Symantec Class 3 Secure Server CA is used to sign certificates for domains belonging to Amazon, Twitter, and Yahoo, which have different domains for different popular apps and in-app APIs.

7.3 Self-signed Certificates

Mobile apps can perform their own certificate verification, which allows them to use certificates not signed by a system-trusted CA. Some of the certificates we encounter are signed by unknown CAs or are self-signed. We see a total of 166 self-signed certificates that are not known trusted root CA certificates. Notable examples of apps using these certificates include apps created by the Russian game developer Zeptolab [100] which use the internal Zeptolab CA. Similarly, some Samsung apps use self signed certificates to communicate with Samsung's push notification servers, and apps made by the Russian technology company Yandex [99] use an internal Yandex CA. While this practice requires these apps to either pin their server certificates in the app or bundle their own CA certificates, some apps simply do not perform the necessary checks to ensure the validity of certificates and to properly authenticate servers.

Lumen cannot determine if these apps use proper certificate pinning: we do not replace self-signed certificates with different self-signed certificates as this might break applications performing pinning. We also cannot replace self-signed certificates with certificates signed by Lumen, as Lumen signed certificates are system trusted (which these self-signed certificates are not); performing these substitutions could open up users to MITM attacks.

Other examples of apps using self-signed certificates include apps used to control Chromecast and Android TV devices that communicate with those devices on the local network. Chromecast certificates have a random UID as Common Name, while Android TV certificates have Common Name strings that include the TV's internal code name, device name, and an ID value (anonymized here as `xxxxx`):

`atvremote/fugu/fugu/Nexus Player/xxxxx`

7.4 Certificate Pinning and CA Bundling

We analyze whether mobile applications or their third-party libraries use certificate pinning or internal CA certificate stores to protect themselves against TLS interception attacks. If an application uses certificate pinning or an internal CA store, it will abort the proxy connection by sending a TLS alert, which causes the Android TLS library on the proxy-side to throw an exception, which contains the error strings generated by the Android C library representing the cause of failure. We mapped those exceptions to their actual cause by replicating each scenario in an app under our control and testing the APIs. We focus on detecting two specific Java exceptions that reveal more interesting aspects of the apps: `CERTIFICATE_UNKNOWN` (*i.e.*, the app implements certificate pinning) and `UNKNOWN_CA` (*i.e.*, the app performs CA bundling). We exclude other Java exceptions related with other TLS events (*e.g.*, due to TLS version incompatibilities, timeouts, or use of expired certificates) as they are less relevant to the apps. We limit our analysis to apps running on devices with an Android SDK version <24 (Android 7.0) as from that version Google has made it mandatory for apps to specify that they want to trust user-installed CA certificates like the ones used by Lumen (§ 2). We furthermore limit our analysis to applications for which we have at least three recorded exceptions, to prevent short-term problems (*e.g.*, CA certificate removal during proxy operation, *etc.*) to show up in our analysis and to minimize the false positive rate. Thus the numbers below are lower-bound approximations.

Cert. Pinning and CA Bundling by Applications: Among the 150 apps that implement certificate pinning, we find 16 pre-installed Google services along with apps like Twitter, Facebook, Dropbox, Instagram, WhatsApp, Uber, 32 finance apps including PayPal family of apps (*e.g.*, Venmo, Braintree, and PayPal), and banking apps like Chase, Capital One, Bank of America, and BBVA. We also found apps like Facebook, Skype, Google apps (*e.g.*, Photos and Voice), and 7 finance/banking apps (*e.g.*, BillMo [14]), implementing CA bundling. However, we did not observe either CA bundling or certificate pinning in 271 other finance/banking apps.

Cert. Pinning and CA Bundling by Third-party Services: The research literature provides ample evidence of private information leakage by mobile advertising and tracking services [39, 78, 79, 90]. Therefore, these services have an incentive to prevent TLS interception which can expose their activities. For example, Crashlytics [22], a tracking and analytics library owned by Google and used by thousands of apps in our dataset, provides support for certificate pinning in their SDK [23]. However, only 9 apps, including Uber and Twitter,

Scenario	Cert. pinning	CA bundling
# apps connecting to first parties	150 [2.0%]	52 [0.7%]
# apps connecting to third parties	94 [1.3%]	20 [0.3%]

Table 2: Number of apps pinning certificates or bundling CAs when connecting to first- and third-party destinations. Total number of apps: N **=7,258**

enabled this feature. We have identified 86 apps pinning certificates when connecting to 15 third-party services including Flurry [40], Localytics [64], and Facebook Graph API [37]. Table 2 summarizes the results of our analysis.

7.5 Other Observations

Certificates have a finite validity duration and can be trusted during that time. However, we observe 11 instances of unique applications using expired certificates. This includes popular apps like AliExpress Shopping App [1] and Any.do [9] who use expired certificates when connecting to their own domains.

8 RELATED WORK

A significant fraction of previous research efforts used passive [56] and active measurements and network scanners like Zmap [32] to make steady progress in understanding the PKI ecosystem [27, 30, 34, 84, 84, 91] and the adoption of SMTP (e-mail) security extensions such as STARTTLS and SPF [28]. Active measurements also allowed studying the scale and impact of TLS vulnerabilities like Heartbleed [29], the presence of RSA keys in end-hosts [52] and how online services and vendors performed certificate revocation in the aftermath of the vulnerability disclosure [51, 63, 101]. Active measurements via online advertising campaigns have also studied TLS proxy deployment in the wild [54, 75]. This study, instead, analyzes Android TLS client behavior.

Mobile TLS and Android TLS Support: Previous studies addressing the use of TLS by Android apps and Android's native TLS support only scratched the surface of the problem. Vallina-Rodriguez *et al.* [89] revealed bloated root stores on devices from specific OS vendors, including ISP-sponsored mobile devices. Georgiev *et al.* [41] used white and black box techniques to detect vulnerabilities in the certificate validation logic in a small number of Android apps and third-party libraries like Google's AdMob [43] and PayPal.

Fahl*et al.* [38] used static analysis on 13,500 popular free Android apps and found that 8% of apps are vulnerable to MITM attacks. Due to the inherent limitations of static analysis, they also manually tested 100 apps using a dedicated testbed with an in-path MITM proxy. Our study increases the number of studied apps in real network conditions by 80x, thus providing a more global picture of TLS use by Android apps. Durumeric *et al.* [31] used CH fingerprints to measure traffic from major browsers and popular interception products. Finally, Wei *et al.* [95] surveyed HTTPS implementation by Android apps and previous research studies, and found significant differences between the theoretical usage of HTTPS and its implementation by app developers, due to improper developer practices, server misconfiguration, lacking documentation, flaws in libraries, the complexity of the TLS PKI system, and lack of end-user knowledge.

9 DISCUSSION

So far, this study has provided a complex and nuanced picture of the diverse (and sometimes incorrect) ways in which Android apps use TLS. We now discuss the implications of our findings, proposing solutions to mitigate the issues found, and discussing Google's role as the platform provider.

Outdated OS TLS Library: Our study revealed that the vast majority of Android apps rely on default OS libraries, using the default configuration to establish TLS connections. As a result, apps are limited to the security guarantees of the OS version that they run on. This indicates that most developers trust the OS to provide them with a secure library with secure default configurations. While this approach is reasonable for most use-cases, it does not guarantee security unless the OS itself stays up to date. In Android, the TLS library is part of the operating system and can only be updated with an OS update. This is especially concerning given that 84% of app-versions use Android's default TLS libraries and real-world Android version adoption data released by Google suggests 61.7% of devices still run old and outdated versions of Android (*e.g.*, Android 5.0 and older).

We propose two remedies: The optimal case would be for Google to devise and enforce a security update policy for *all* device vendors, which obligates them to update their devices with the latest security patches in a timely fashion for a reasonable amount of time after release. The alternative is to separate the TLS libraries from the core OS so that they can be updated separately. Google has taken this approach before, *e.g.*, with Google Play Services which allows apps to access Google APIs such as Maps and Drive.

CA Certificates and Trust Stores: Trusting CA certificates present in the OS-provided trust store has many of the downsides of using OS-default libraries: updates (addition of new CA certificates, revocation of untrusted CAs, *etc.*) are also tied to OS updates. Moreover, our previous work [89] has revealed substantial differences across Android trust stores: devices produced by certain vendors contain certificates that are not present elsewhere, some of which are dubious CA certificates that could be used to perform MITM attacks by an in-path adversary. As a result, some apps resort to pinning certificates and bundling their own CA trust stores which, when done correctly, is a good security measure against MITM attacks but can be problematic when implemented poorly. Apps that use these methods need to implement their own validation and verification methods, as there are no default API calls to do so in those circumstances, which can in turn cause apps to be less secure due to developer inexperience. This practice is not common among apps in our data (< 2% of apps), but we have observed examples of its poor implementation: *e.g.*, a popular recovery management app with root privileges downloads a CA bundle in the clear, making it vulnerable to MITM attacks. [5]

Control Over Library Settings: Android only allows a limited amount of customization of TLS settings. Apps can set the SNI, the list of supported cipher suites, and supported protocol versions.

[5] We reached out to the developers of the app to notify them of the vulnerability in June 2017, but have not heard back as of October 2017.

While this covers the needs of many apps, there are apps that require setting TLS features that are not exposed by the Android API. It currently is, for example, impossible to set the ALPN extension value although it is supported by Android's internal TLS library. As a result, apps that need to negotiate a specific ALPN during handshake (*e.g.*, Facebook), are essentially forced to bundle a different TLS library altogether. The app developer will have to keep their bundled TLS library up-to-date and ensure its proper configuration, as failing to do so will make their TLS communications vulnerable. Giving more configuration and control options to app developers, especially for TLS extensions, would remove the need of apps to bundle TLS libraries and potentially becoming less-secure as a result.

Education and Guidelines: TLS is a complex protocol with many different parameters that need to be considered carefully in order to make TLS communication secure. App developers need to be educated on how to configure and use it correctly. The availability of a default API with reasonable and secure default configuration helps, but without implementation guidelines and security best-practices, app developers who are less familiar with the protocol are more likely to unintentionally make poor decisions. To aid app developers in securing their TLS flows, Google has released an open source tool for TLS testing and troubleshooting [73].

Additionally, even with up-to-date and configurable default libraries, it is inevitable that some developers will bundle other TLS libraries with their apps and therefore have to stay up-to-date on library versions and vulnerability mitigations. Companies like Microsoft and Facebook who can dedicate sufficient resources to security are more likely to maintain secure use of third-party TLS libraries, while smaller developers might not be able to do so. In the past Google has issued a security advisory warning developers about using a vulnerable version of GnuTLS and deployed a detection algorithm to warn developers about its presence in their apps. That, combined with their introduction and use of GREASE to ensure proper server-side TLS implementation and ease of protocol expansion in the future, are steps in the right direction showing their willingness to improve the state of TLS in Android, but more work is needed.

10 CONCLUSIONS AND FUTURE WORK

There has been an increase in concerns over privacy, security, and encryption in recent years; and with emergence of reports of widespread Internet surveillance and attacks on user privacy, there has been a push to encrypt all Internet traffic. That, combined with the surge in use of mobile apps for sensitive applications (*e.g.*, banking and e-commerce) and purposes that go beyond traditional web services (*e.g.*, P2P communications and IoT activities) in the past decade, has resulted in the increasing reliance of apps on TLS for encrypted online communications, making it imperative to ensure its implementation in mobile apps remains secure.

We demonstrated that it is possible to identify most mobile TLS traffic by the cipher suite list, providing an unprecedented view of TLS usage in Android in the wild, and at scale, using data collected by Lumen, which we have made available to the public. We showed that most apps use default TLS libraries, and are vulnerable to attacks resulting from running on outdated Android OS versions. We also uncovered diverse ways in which apps use default and third-party TLS libraries, and found examples of widely-used apps with alarming vulnerabilities and weaknesses in their implementation, including announcement of null ciphers and anonymous key exchange methods. We found certificate pinning and CA bundling to be almost testimonial and mainly limited to apps developed by larger companies, and showed that the use of TLS in apps can change over time by providing examples of popular apps evolving over different versions.

We concluded our study with a discussion on the state of Android TLS, and provided suggestions to improve the status quo and remedy some of the issues associated with Android's implementation of TLS. As future work, we plan to expand our methods in order to study apps' resilience to different types of attacks, and to test their quality of certificate verification via active measurements. We would also like to combine our data with active server-side measurements to explore how mobile app servers are configured to use TLS, and whether we see a difference between them and servers serving web content or other services leveraging TLS.

11 ACKNOWLEDGMENTS

This project is funded by the NSF grants CNS-1564329, CNS-1528156, CNS-1350720, and the Data Transparency Lab Grants (2016). The authors would like to acknowledge Eduardo Acha and Prof. Alejandro Hevia (Universidad de Chile) for sharing their list of vulnerable cipher suites. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not reflect the views of the funding bodies.

REFERENCES

[1] AliExpress Shopping App 2017. (2017). https://play.google.com/store/apps/details?id=com.alibaba.aliexpresshd
[2] ALPN Status 2017. https://github.com/http2/http2-spec/wiki/ALPN-Status. (2017).
[3] S. Anand, M. Naik, M.J. Harrold, and H. Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proc. of the International Symposium on the Foundations of Software Engineering (FSE)*.
[4] Android 6.0 Changes 2015. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html. (2015).
[5] Android Developer Reference: KeyChain 2017. https://developer.android.com/reference/android/security/KeyChain.html. (2017).
[6] Android Developer Reference: KeyStore 2017. https://developer.android.com/reference/java/security/KeyStore.html. (2017).
[7] Android Developer Training – Security with HTTPS and SSL 2017. https://developer.android.com/training/articles/security-ssl.html. (2017).
[8] Android Explained: Android 5.1 Changelog 2017. https://www.androidexplained.com/android-5-1-changelog/. (2017).
[9] Any.do 2017. https://play.google.com/store/apps/details?id=com.anydo. (2017).
[10] Appsflyer 2017. https://www.appsflyer.com/. (2017).
[11] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall. 2013. Staying Online While Mobile: The Hidden Costs. In *Proc. ACM Int. Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.
[12] Barker, Elaine and Roginsky, Allen. 2015. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. NIST Special Publication 800-131A. (2015). http://dx.doi.org/10.6028/NIST.SP.800-131Ar1
[13] K. Bhargavan and G. Leurent. 2016. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *Proc. Network and Distributed System Security Symposium (NDSS)*.
[14] BillMo Money Transfer and Wallet 2017. https://play.google.com/store/apps/details?id=com.billmo.android. (2017).
[15] BlackBerry Limited. 2017. BBM - Free Calls and Messages. https://play.google.com/store/apps/details?id=com.bbm. (2017).
[16] D. Boneh, S. Inguva, and I. Baker. 2017. SSL Man in the Middle Proxy. https://crypto.stanford.edu/ssl-mitm/. (2017).
[17] BoringSSL 2017. https://boringssl.googlesource.com/boringssl/. (2017).

[18] Changes to Trusted Certificate Authorities in Android Nougat 2016. (2016). https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html
[19] J. Clark and P. van Oorschot. 2013. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proc. IEEE Symposium on Security and Privacy (S&P)*.
[20] S. Comino, F. M. Manenti, and F. Mariuzzo. 2016. Updates Management in Mobile Applications. iTunes vs Google Play. CCP Working Paper.
[21] C. Conlon. 2011. Installing an Alternate SSL Provider on Android. http://www.linuxjournal.com/article/10896. *Linux Journal* 2011, 205 (2011), 5.
[22] Crashlytics 2017. http://www.crashlytics.com. (2017).
[23] Crashlytics API 2017. Crashlytics API. https://docs.fabric.io/javadocs/crashlytics/2.3.2/com/crashlytics/android/Crashlytics.html. (2017).
[24] D. Benjamin. 2016. Applying GREASE to TLS Extensibility. IETF Draft. (2016). https://tools.ietf.org/html/draft-davidben-tls-grease-01
[25] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Standards Track). (2008). https://tools.ietf.org/html/rfc5246
[26] Dolphin Browser 2017. https://play.google.com/store/apps/details?id=mobi.mgeek.TunnyBrowser. (2017).
[27] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J.A. Halderman. 2015. A Search Engine Backed by Internet-Wide Scanning. In *Proc. ACM Conference on Computer and Communications Security (CCS)*.
[28] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J.A. Halderman. 2015. Neither Snow Nor Rain Nor MITM...: An Empirical Analysis of Email Delivery Security. In *Proc. ACM Int. Measurement Conference (IMC)*.
[29] Z. Durumeric, J. Kasten, D. Adrian, J.A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. 2014. The Matter of Heartbleed. In *Proc. ACM Int. Measurement Conference (IMC)*.
[30] Z. Durumeric, J. Kasten, M. Bailey, and J.A. Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *Proc. ACM Int. Measurement Conference (IMC)*.
[31] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J.A. Halderman, and V. Paxson. 2017. The Security Impact of HTTPS Interception. In *Proc. Network and Distributed System Security Symposium (NDSS)*.
[32] Z. Durumeric, E. Wustrow, and J.A. Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications.. In *Proc. USENIX Security Symposium*, Vol. 2013.
[33] EA Games 2017. (2017). http://www2.ea.com/android
[34] EFF. 2017. The EFF SSL Observatory. https://www.eff.org/observatory. (2017).
[35] N. Elenkov. 2014. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press.
[36] F-Secure Freedome VPN 2017. (2017). https://play.google.com/store/apps/details?id=com.fsecure.freedome.vpn.security.privacy.android
[37] Facebook Graph API 2017. https://developers.facebook.com/docs/graph-api. (2017).
[38] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proc. ACM Conference on Computer and Communications Security (CCS)*.
[39] Federal Trade Commission. 2016. https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked. (2016).
[40] Flurry. Yahoo Mobile Developer Suite 2017. http://www.flurry.com/. (2017).
[41] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proc. ACM Conference on Computer and Communications Security (CCS)*.
[42] GnuTLS Transport Layer Security Library 2017. http://www.gnutls.org/. (2017).
[43] Google. 2017. AdMob by Google. https://www.google.com/admob/. (2017).
[44] Google. 2017. Google Analytics. https://analytics.google.com. (2017).
[45] Google. 2017. HTTPS at Google. (2017). https://www.google.com/transparencyreport/https/
[46] Google. 2017. Nexus Help: Check & update your Android version. (2017). https://support.google.com/nexus/answer/4457705
[47] Google Cloud Messaging 2017. Google Cloud Messaging. http://developer.android.com/google/gcm/index.html. (2017).
[48] Google Security Blog. 2014. (2014). https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html
[49] M. Green. 2017. Attack of the week: FREAK (or factoring the NSA for fun and profit). (2017). https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/
[50] gRPC. 2017. A high performance, open-source universal RPC framework. (2017). http://www.grpc.io/
[51] M. Hastings, J. Fried, and N. Heninger. 2016. Weak Keys Remain Widespread in Network Devices. In *Proc. ACM Int. Measurement Conference (IMC)*.
[52] N. Heninger, Z. Durumeric, E. Wustrow, and J.A. Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.. In *Proc. USENIX Security Symposium*.
[53] Hola VPN 2017. (2017). https://play.google.com/store/apps/details?id=org.hola
[54] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. 2014. Analyzing Forged SSL Certificates in the Wild. In *Proc. IEEE Symposium on Security and Privacy (S&P)*.
[55] IANA. 2017. Transport Layer Security Parameters. (2017). https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml
[56] ICSI 2017. ICSI Certificate Notary. https://notary.icsi.berkeley.edu. (2017).
[57] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson. 2016. An Analysis of the Privacy and Security Risks of Android VPN Permission-enabled Apps. In *Proc. ACM Int. Measurement Conference (IMC)*.
[58] Jio4GVoice 2017. https://play.google.com/store/apps/details?id=com.jio.join. (2017).
[59] K-9 Mail 2017. https://play.google.com/store/apps/details?id=com.fsck.k9. (2017).
[60] K. Conger. 2016. Apple will require HTTPS connections for iOS apps by the end of 2016. (2016). https://techcrunch.com/2016/06/14/apple-will-require-https-connections-for-ios-apps-by-the-end-of-2016/
[61] D. Karrenberg, Y. Rekhter, E. Lear, and G. Jan de Groot. 1996. Address Allocation for Private Internets. RFC 1918. (Feb. 1996). https://doi.org/10.17487/rfc1918
[62] A. Langley, A. Pironti, R. Barnes, and M. Thomson. 2015. Deprecating Secure Sockets Layer Version 3.0. *RFC7568* (2015).
[63] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson. 2015. An end-to-end measurement of certificate revocation in the web's PKI. In *Proc. ACM Int. Measurement Conference (IMC)*.
[64] Localytics 2017. http://www.localytics.com/. (2017).
[65] Lumen Privacy Monitor 2017. https://play.google.com/store/apps/details?id=edu.berkeley.icsi.haystack. (2017).
[66] Man in the Middle 2017. Man in the Middle Proxy. http://mitmproxy.org. (2017).
[67] B. McMahan and D. Ramage. 2017. Federated Learning: Collaborative Machine Learning without Centralized Training Data. (2017). https://research.googleblog.com/2017/04/federated-learning-collaborative.html
[68] Miniclip 2017. https://www.miniclip.com/. (2017).
[69] MoboGenie 2017. http://www.mobogenie.com/. (2017).
[70] B. Moeller and A. Langley. 2015. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (Proposed Standard). (April 2015), 8 pages. https://doi.org/10.17487/RFC7507
[71] Mozilla. 2017. Network Security Services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS. (2017).
[72] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proc. ACM Int. Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.
[73] Nogotofail 2017. Nogotofail – An on-path blackbox network traffic security testing tool. https://github.com/google/nogotofail. (2017).
[74] M. O'Connor. 2014. Google Services Updated to Address OpenSSL CVE-2014-0160 (the Heartbleed bug). (2014). https://security.googleblog.com/2014/04/google-services-updated-to-address.html
[75] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. 2016. TLS proxies: Friend or foe?. In *Proc. ACM Int. Measurement Conference (IMC)*.
[76] Popov, A. 2015. Prohibiting RC4 Cipher Suites. (2015). https://tools.ietf.org/html/rfc7465
[77] Qualys. 2017. SSL Labs' database of user agent capabilities. (2017). https://www.ssllabs.com/ssltest/clients.html
[78] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. 2015. Haystack: In Situ Mobile Traffic Analysis in User Space. *ArXiv e-prints* (2015).
[79] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proc. ACM Int. Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[80] Rovio 2017. https://www.rovio.com. (2017).
[81] Samsung Internet Browser 2017. https://play.google.com/store/apps/details?id=com.sec.android.app.sbrowser. (2017).
[82] SSL Labs 2017. https://www.ssllabs.com. (2017).
[83] Super Mario Run 2017. https://play.google.com/store/apps/details?id=com.nintendo.zara. (2017).
[84] P. Švenda, M. Nemec, P. Sekan, R. Kvašňovský, D. Formánek, D. Komárek, and V. Matyáš. 2016. The Million-Key Question–Investigating the Origins of RSA Public Keys. In *Proc. USENIX Security Symposium*.
[85] Syncthing. 2017. Block Exchange Protocol v1. (2017). https://docs.syncthing.net/specs/bep-v1.html
[86] D. Thomas, A. Beresford R, and A. Rice. 2015. Security Metrics for the Android Ecosystem. In *Proc. of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*.
[87] Tiffany Alvord DreamWorld 2017. https://play.google.com/store/apps/details?id=com.stargirlapps.google.tiffany. (2017).
[88] TuneIn Radio 2017. https://play.google.com/store/apps/details?id=tunein.player. (2017).

[89] N. Vallina-Rodriguez, J. Amann, C. Kreibich, N. Weaver, and V. Paxson. 2014. A Tangled Mass: The Android Root Certificate Stores. In *Proc. ACM Int. Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.
[90] N. Vallina-Rodriguez, S. Sundaresan, A. Razaghpanah, R. Nithyanand, M. Allman, C. Kreibich, and P. Gill. 2016. Tracking the Trackers: Towards Understanding the Mobile Advertising and Tracking Ecosystem. In *Proc. of the Workshop on Data and Algorithmic Transparency (DAT)*.
[91] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J.A. Halderman. 2016. Towards a Complete View of the Certificate Ecosystem. In *Proc. ACM Int. Measurement Conference (IMC)*.
[92] Viber 2017. https://play.google.com/store/apps/details?id=com.viber.voip. (2017).
[93] VLC for Android 2017. https://play.google.com/store/apps/details?id=org.videolan.vlc. (2017).
[94] WebRTC 2017. https://webrtc.org/start/. (2017).
[95] X. Wei and M. Wolf. 2016. A Survey on HTTPS Implementation by Android Apps: Issues and Countermeasures. *Applied Computing and Informatics* (2016).
[96] Wire - Private Messenger 2017. https://play.google.com/store/apps/details?id=com.wire. (2017).
[97] Wired. 2017. Half the Web is Now Encrypted. That Makes Everyone Safer. (2017). https://www.wired.com/2017/01/half-web-now-encrypted-makes-everyone-safer/
[98] M.Y. Wong and D. Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *NDSS*.
[99] Yandex 2017. https://www.yandex.com/. (2017).
[100] ZeptoLab 2017. https://www.zeptolab.com/. (2017).
[101] L. Zhang, D. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, and C. Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *Proc. ACM Int. Measurement Conference (IMC)*.