# A Comparative Analysis of Certificate Pinning in Android & iOS

Amogh Pradeep*
Northeastern University
USA

Muhammad Talha Paracha*
Northeastern University
USA

Protick Bhowmick
Virginia Tech
USA

Ali Davanian
University of California, Riverside
USA

Abbas Razaghpanah
ICSI / Cisco Inc.
USA

Taejoong Chung
Virginia Tech
USA

Martina Lindorfer
TU Wien
Austria

Narseo Vallina-Rodriguez
IMDEA Networks / AppCensus Inc.
Spain

Dave Levin
University of Maryland
USA

David Choffnes
Northeastern University
USA

## ABSTRACT

TLS certificate pinning is a security mechanism used by applications (apps) to protect their network traffic against malicious certificate authorities (CAs), in-path monitoring, and other methods of TLS tampering. Pinning can provide enhanced security to defend against malicious third-party access to sensitive data in transit (*e.g.,* to protect sensitive banking and health care information), but can also hide an app's personal data collection from users and auditors. Prior studies found pinning was rarely used in the Android ecosystem, except in high-profile, security-sensitive apps; and, little is known about its usage on iOS and across mobile platforms.

In this paper, we thoroughly investigate the use of certificate pinning on Android and iOS. We collect 5,079 unique apps from the two official app stores: 575 common apps, 1,000 popular apps each, and 1,000 randomly selected apps each. We develop novel, cross-platform, static and dynamic analysis techniques to detect the usage of certificate pinning. Thus, our study offers a more comprehensive understanding of certificate pinning than previous studies.

We find certificate pinning as much as 4 times more widely adopted than reported in recent studies. More specifically, we find that 0.9% to 8% of Android apps and 2.5% to 11% of iOS apps use certificate pinning at runtime (depending on the aforementioned sets of apps). We then investigate which categories of apps most frequently use pinning (e.g., apps in the "finance" category), which destinations are typically pinned (e.g., first-party destinations vs those used by third-party libraries), which certificates are pinned and how these are pinned (e.g., CA vs leaf certs), and the connection security for pinned connections vs unpinned ones (e.g., the use of weak ciphers or improper certificate validation). Lastly, we investigate how many pinned connections are amenable to binary instrumentation to reveal the contents of their connections; for those that are, we analyze the data sent over pinned connections to understand what is protected by pinning.

## CCS CONCEPTS

• **Security and privacy → Network security**; *Web protocol security*; **Mobile and wireless security**.

## KEYWORDS

Certificate Pinning, App Pinning, Transport Layer Security, TLS, Network Security, Measurement Techniques

**ACM Reference Format:**
Amogh Pradeep, Muhammad Talha Paracha, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. 2022. A Comparative Analysis of Certificate Pinning in Android & iOS. In *ACM Internet Measurement Conference (IMC '22), October 25–27, 2022, Nice, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3517745.3561439

## 1 INTRODUCTION

Mobile (applications) apps are extremely popular – 230 billion apps were installed on devices in 2021 [39] alone – and often transmit sensitive data over the Internet to deliver their service (*e.g.,* credentials, financial and health information). Thus, network connection security is critically important in this context. While the standard TLS PKI provides sufficient security for most apps, several classes of attacks have revealed gaps in its protection: tampered, misconfigured, or poorly maintained certificate authority (CA) root stores [42] can enable highly targeted or large-scale monkey-in-the-middle (MITM) attacks [6, 14]. To address this problem, app developers and third-party libraries can use *certificate pinning*, which establishes a developer-specified relationship between a hostname and its cryptographic identity (certificate or hash of the public key)—one that is typically hard-coded (hence "pinned") and that adds another layer of security compared to certificate validation that uses only the trusted system CA root store.

*Equal contribution.


*IMC '22, October 25–27, 2022, Nice, France*

ACM ISBN 978-1-4503-9259-4/22/10…$15.00
https://doi.org/10.1145/3517745.3561439

Although beneficial from a security standpoint, pinning is known to introduce maintenance overheads, misconfiguration errors, and other problems which could expose users to more attacks. Unfortunately, there is no clear community consensus on whether the benefits of pinning outweigh potential risks of misconfiguration and developer errors. On the web, it was first introduced in 2011 [9], but has since been deprecated by all major desktop and mobile browsers [18]. Android officially supports pinning since version 4.2 (released in 2012) [24], but has since moved to not recommending pinning due to the risk of app breakage when server configurations change [15, 25]. Apple does not provide clear recommendations for iOS, but notes that pinning might be necessary to meet regulatory requirements [17], and recommends long-term strategies to handle certificate changes. More generally, OWASP, a community-driven approach to establish app security testing standards and guidelines, advocates for pinning [31] against sophisticated attacks in its Mobile Application Security Verification Standard, which is frequently used as a basis for app security audits. It is, therefore, vital to know whether app developers implement certificate pinning; and to identify common deployment errors that could compromise apps' security.

In this work, *we provide the first multi-perspective look at certificate pinning, by developing more complete methodologies for detecting it that leverage the complementary strengths of static and dynamic analysis, characterizing its prevalence across iOS and Android, and investigating the implications of observed implementations.* We develop novel static and dynamic techniques to detect and measure the adoption of pinning. Specifically, our methodology includes more complete rules for searching app binaries for evidence of certificates or pinning APIs, an analysis of which code is responsible for pinning, as well as run-time analysis that reliably distinguishes pinned connections from other confounding types of TLS connection behavior. Our work builds upon and extends prior work in this space [21, 30, 33, 34], as we discuss in detail in Section 2.2.

We run our detection techniques on multiple datasets of apps, allowing us to understand the prevalence of certificate pinning according to various criteria. These datasets consist of 575 apps present on both platforms, 1,000 popular apps on each platform, and a random selection of 1,000 apps from each of their respective main app stores, the Play Store and the App Store (5,079 unique apps in total). Our analysis pipeline allows us to identify that pinning is not at all negligible: 6.7% (Android) and 11.4% (iOS) of popular apps use pinning at run time. Further, we find substantial differences between platforms when it comes to the prevalence of pinning, the consistency of pinning for the same app in iOS and Android, and the connection security of those pinned apps. This comparative analysis allows us to investigate whether developers and third-party SDK providers systematically use pinning as a security mechanism across their products, regardless of the platform.

We then investigate which categories of apps most frequently use pinning (apps in the "finance" category), which destinations are typically in pinned connections (first-party destinations vs those used by third-party libraries), which certificates are pinned and how they are pinned (CA vs leaf certs), and the connection security for pinned connections vs unpinned ones (e.g., the use of weak ciphers or improper certificate validation). Last, we investigate how many pinned connections are amenable to binary instrumentation for revealing the contents of their connections, and for those that are, we analyze the data sent in pinned connections to understand what is protected by pinning.

To summarize our key results:

- We find a wide range of prevalence for potential and actual pinning, with 11.4% of popular iOS apps and 6.7% of popular Android apps using pinning in our dynamic tests. Static analysis reveals even more potential pinning (up to 27% on Android and 33% on iOS). Pinning is much less prevalent in randomly selected (*i.e.,* less popular) apps (0.9% of Android and 2.5% of iOS apps).
- Of the 27 apps that pin on both iOS and Android, fewer than half (13) do so consistently across platforms.
- If an app uses pinning, it does so selectively, usually on a small fraction of the domains contacted. The majority of destinations that are pinned are third-party sites. This finding is also substantiated by our code analysis, in which we find pinning most commonly in third-party libraries (social networks, payment processing, and app analytics).
- When certificates are pinned, the vast majority are certificates using the default PKI (as opposed to a custom one) *i.e.,* the leaf is signed (directly or indirectly) by a CA available in one or more public root stores. Nearly three quarters of the pinned certificates are CA certificates, with the remaining being leaf certificates.
- For the connections for which we could disable pinning, we found no statistically significant increase in PII compared to non-pinned connections, with the exception of Advertiser IDs on iOS. This indicates that pinning is not typically used to protect (non-credential) PII, or to hide the collection of such information.

To support reproducibility and facilitate further research in the area, we make our dataset and code publicly available at: https://github.com/NEU-SNS/app-tls-pinning.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Definition of Pinning

During a TLS handshake, clients obtain a *certificate chain* (ordered list of certificates) from servers, where each certificate is signed by the previous one. Clients trust the chain if they trust the *root* certificate, and the signatures from the *root* (first) to the *leaf* (last) are all valid. A *root store* or *CA (certificate authority) store* is a collection of such trusted *root* certificates, which is included in OSes including Android and iOS [28, 42].

Certificate pinning is an alternate to trusting OS root certificates, where applications include a custom certificate to be trusted (in source code), instead of the set of certificates present on the OS. We define pinned certificates as such custom certificates that *must* be present in the certificate chain to successfully establish a TLS connection. These pinned certificates could be any certificate in the chain, *i.e.,* leaf, intermediate, or root certificates. They could also be pinned in any form, *i.e.,* storing the entire certificate, a hash of the certificate, or some other identifier.

**Pinning for Protection:** Mobile root stores are known to include expired, unknown, or obscure CA certificates [42], which can expose clients to TLS interception attacks. An attacker with access to the private key for a CA certificate in the system trust store can use it to sign arbitrary certificates (for arbitrary domains) and trick

the client into accepting these malicious certificates as valid. Using certificate pinning prevents such attacks by limiting certificate trust to a pre-determined set of certificates instead of trusting a certificate issued by any CA certificate in the system trust store. Note that certificate pinning not only protects against malicious actors, but also against investigators and auditors seeking to analyze the data exchanged between devices and servers (*e.g.,* to understand personal data exfiltration, cross-border data transfers, *etc.*).

**Pinning for Customization:** Certificate pinning enables developers to define a specific certificate to trust. This allows developers to issue and sign their own trusted certificates instead of obtaining one from a trusted third-party CA, thus regaining more control over their internal certificates at the cost of limited utility since custom CAs will not be trusted by browsers or other software that does not trust the custom CA.

Note, however, that verifying if a pinned certificate is present in a chain is not sufficient to ensure that the chain is correct; rather, the TLS library must still validate all other properties of certificates (*i.e., Common Name* matching, revocation checking, *etc.*) to protect against various other attacks.

**Pinning and HPKP:** We note that certificate pinning methods found in mobile apps differ greatly from *HTTP Public Key Pinning* (HPKP). HPKP is an obsolete technique for *web browsers* that allowed website owners to specify pinned certificates for their domain. One key reason HPKP was proposed is that website owners in general cannot directly control the trust store for a browser, and HPKP gave them a way to specify custom certificates for pinning on a domain. In contrast, mobile services that use pinning can control both the client software (the app) and the web servers they communicate with. As such, there is no need for any additional protocol like HPKP to specify how pinning should occur—mobile apps simply include the pinned certificate material in the app code and/or metadata.

We also note that the threat models and stakeholders in the two techniques are different. For HPKP, the website owner does not trust the OS or browser root store, but assumes that browser will enforce a specified pinned certificate. Further, HPKP trusts the first seen certificate (and thus does not solve the problem for adversaries that can intercept the first TLS connection) and also does not support changing the pinned certificate. In contrast, mobile services that use certificate pinning do not trust the OS root store, but trust that the OS will faithfully execute its specified certificate validation and pinning code. In addition, mobile services can change the pinned certificate in numerous ways, *e.g.,* by pinning a CA certificates that can issue additional trusted leaf certificates, or releasing a new version of the app with a new pinning specification.

## 2.2 Related Work

**TLS Studies:** Georgiev *et al.* [23] conducted one of the earliest studies about TLS usage in non-browser software on multiple platforms including Android and iOS. They found instances of insecure TLS implementations on both Android and iOS, but did not find any evidence of certificate pinning. An early study of TLS usage on Android was conducted by Fahl *et al.* [21] in 2012. They developed static and dynamic analysis techniques to detect insecure TLS implementations, and found 8% of apps to be vulnerable to MITM attacks. While certificate pinning was not their focus, they performed manual dynamic analysis on 20 cherry-picked high profile apps and found 2 instance of pinning.

In 2017, Razaghpanah *et al.* [34] studied TLS usage by Android apps using on-device traffic monitoring. They found 150 apps in their dataset that implemented some form of pinning (2% of apps analyzed). Further, they investigated pinning behavior in first and third-party destinations, and made observations about the type of apps that use pinning (*i.e.,* social and finance apps). The key differences between their work and ours are that they (a) study a single platform using a single technique (dynamic analysis), and (b) use an uncontrolled set of apps, sourced from 5000 users who downloaded the Lumen app and consented to their analysis.

Stone *et al.* [40] conducted a TLS study closely related to our work. They studied 400 security-sensitive Android and iOS applications, and presented a dynamic analysis technique to detect pinned connections that fail to validate certificate hostnames. However, their technique only finds apps that pin intermediate or root certificates in the certificate chain. In contrast, our dynamic and static analysis techniques cover all pinned certificates.

**Android Network Security Configurations:** In 2015, Android 6.0 introduced Network Security Configurations (NSCs) [20] that enable apps to customize certain network security settings. Oltrogge *et al.* [30] studied NSC adoption using static analysis and found that 7.43% of 1.3M apps used NSCs; with only 0.67% (of the 7.43%) using pinning. The authors reported pinning to be most common in apps from the "finance" category, and that majority of pinned certificates (≈63%) were root CAs (rather than leaf certificates). In this work, we revisit such characteristics and find many similarities in our results. Although the authors primarily employed static analysis, they also performed manual dynamic analysis on a small set of 40 apps to discover potential PII usage in plaintext HTTP connections. Possemato *et al.* [33] studied Network Security Policies (a single line configuration in Android Manifests, or NSC files) and found that 13.02% of 125k+ apps used these policies; with only 0.62% (of the 13.02%) using pinning. Interestingly, they found instances of misconfigurations related to pinning (e.g., *example.com* pinned, *overridePins* attribute set to "true").

Although NSCs provide a simple way for developers to secure network connections, previous studies have seen their adoption lacking. Since apps can use custom mechanisms to pin certificates, relying solely on standard configuration files (like NSCs) to understand pinning is inadequate. We utilize static NSC analysis in this study, as a way to compare to previous techniques and improve it with novel techniques to detect certificate pinning that have previously been unexplored.

**Pinning Recommendations:** Oltrogge *et al.* [29] devised techniques to evaluate whether pinning is a suitable strategy for an app. After analyzing 600k+ Android apps, they conservatively proposed pinning for 1.8% of them. Although not their primary goal, the authors used semi-automated static code analysis to infer the prevalence of pinning, and found it to be merely ≈0.07% in their dataset. They also surveyed 45 app developers to understand how they perceive pinning—while only a quarter of these developers knew about pinning, all of them found it too complex to use.

**iOS Network Security Mechanisms:** Due to the closed-source nature of iOS, there is limited research on analyzing how iOS apps

| | Android | | | iOS | | |
|---|---|---|---|---|---|---|
| **Rank** | **Random** | **Popular** | **Common** | **Common** | **Popular** | **Random** |
| 1 | Education 12% | Games 36% | Games 18% | Games 18% | Games 21% | Games 15% |
| 2 | Games 12% | Weather 2% | Productivity 12% | Productivity 14% | Photography 11% | Business 11% |
| 3 | Tools 6% | Finance 2% | Business 7% | Business 8% | Social 6% | Education 11% |
| 4 | Music 6% | Shopping 2% | Communication 6% | Social 7% | Education 6% | Food 7% |
| 5 | Books 6% | Entertainment 2% | Finance 6% | Education 6% | Finance 6% | Lifestyle 7% |
| 6 | Business 5% | Food 2% | Education 5% | Finance 6% | Lifestyle 5% | Utilities 6% |
| 7 | Lifestyle 5% | Social 2% | Social 5% | Utilities 5% | Entertainment 4% | Entertainment 4% |
| 8 | Entertainment 4% | Productivity 2% | Health 4% | Photography 4% | Utilities 4% | Health 4% |
| 9 | Travel 4% | Photography 2% | Travel 3% | Health 3% | Productivity 4% | Travel 4% |
| 10 | Personalization* 4% | Music 2% | Lifestyle 3% | Lifestyle 3% | Weather 4% | Shopping 3% |

**Table 1: An overview of our app datasets. We present the top 10 app categories from each dataset, along with their percentages over the total number of apps in that dataset.**

adopt network security mechanisms. Tang *et al.*[41] studied security vulnerabilities in iOS apps that exist at the TCP/IP layer layer due to open ports. Orikogbo *et al.*[32] studied the validity of TLS certificates used by these apps. Both of these works introduce novel ways to build a corpus of iOS apps. Our work complements prior research by using their app collection techniques and exploring another aspect of network security: TLS pinning prevalence and implementations in the iOS ecosystem.

**Android vs iOS:** Besides the limited number of iOS studies mentioned above, by far the majority of related work on security and privacy issues in mobile apps focuses on the Android platform. Even rarer is related work performing comparative studies between both platforms: Han *et al.* [26] compared the usage of security sensitive APIs for 2.6k cross-platform apps in 2013. Chen *et al.* [19] investigated the inclusion of potentially harmful third-party libraries in 1.3M Android and 160K iOS apps in 2016 and found libraries to show similarly risky behavior on both platforms. In the area of privacy, Ren *et al.* [38] performed the first comparison of the PII collected by the 100 most popular Android apps compared to their iOS counterparts. Most recently, Kollnig *et al.* [27] performed a comparative study of PII leakage and tracking in 12k apps from each platform, concluding that neither platform is clearly better at protecting user privacy. However, they considered connection security out of scope and circumvented it (to the extent possible) by disabling default certificate validation by the OS.

## 2.3 Study Goals

Our study is organized around the following key research questions, which we answer in Sect. 5 based on the datasets presented in Sect. 3 and methodology discussed in Sect. 4:

(RQ1) How can we reliably detect pinning and its prevalence in mobile apps, in a platform-agnostic way?
(RQ2) What are the characteristics of apps that deploy pinning (popular vs unpopular apps, app categories, pinned destinations) and what are their implications?
(RQ3) How consistently developers use pinning across the Android and iOS versions of the same apps?
(RQ4) How is pinning implemented (*e.g.,* nature of certificate chains, code that contributes to pinning)?
(RQ5) How secure is pinning in mobile apps? And what kind of data is protected by pinning?

# 3 DATASETS

To understand the prevalence of pinning in different parts of the Android and iOS ecosystems, we collect a wide and diverse range of apps on both platforms. We group the apps in three different datasets: popular apps, random apps, and "common" apps. The "common" apps dataset contains the same app on Android and iOS, thus enabling us to perform head-to-head comparisons of the two platforms. We collect these apps at various points in time in 2021.

Collecting Android apps from the Google Play Store is simpler than collecting iOS apps from the Apple App Store. For Android, we use GPlayCLI [10] to download apps directly from the Play Store. For iOS, we automate GUI interactions with the deprecated iTunes 12.6 application to download apps, based on previous work [32]. For more details about the challenges with iOS app crawls, we refer the reader to Appendix A. We obtain the category of each app (e.g., gaming or finance) directly from the metadata set by the developers and available in the respective stores.

**Common Apps (n = 575):** Linking apps present on one market with those present on another is non-trivial. We create the set of common apps using AlternativeTo [1]. This website crowdsources information, recommendations, and reviews for software. Apps listed on this website can have links to the Google Play Store and Apple App Store if they are present on both platforms. We retrieve ≈1000 app pages sorted by popularity on this website and look for apps listed on both stores. Using this technique, we obtain 575 apps; we manually verify (on a small random sample of 30 apps) that these apps are in fact the same. To respect community norms related to crawling, we add our contact information in the User-Agent field, and limit our crawler to request 1 page per second.

**Popular Apps (n = 1000):** For popular apps on Android, we use the *google-play-scraper* [8] to crawl "Top Free" lists for each category on the Google Play Store. We pick at random 1000 apps from these lists (≈12k in total). For iOS, we use the iTunes Search API to fetch top apps using 19 generic category names as search terms (*e.g.,* productivity, finance, music). The API returns at most 100 results per call. We repeat the process for each category and collect unpaid apps that are compatible with our test device, compiling a set of 1000 apps. Both sets contain apps that capture the notion of

popularity for each store; they do not necessarily represent the top 1000 apps for either platform. We note that we used US version of the app stores while compiling these listings.

**Random Apps (n = 1000):** To compile a list of random apps, we start out with fetching details about as many apps as possible. Unfortunately, the list of all apps present on either platform is not public. For Android, we use a list of 1.35M app IDs compiled by prior work [30]. For iOS, we crawl 1.25M app IDs from the official store listings [2]. From each of these lists, we randomly select 1000 apps and download them from the respective stores. We believe that the large size of our lists provides a sufficient degree of randomness for our analysis.

We perform all crawls from North America; Common and Popular sets were collected from February to May, and Random sets were collected in October of 2021. Due to our app collection technique, we see app collisions between the three sets; in such cases the same app is used in the sets they appear in. Accounting for collisions, we collect 2,564 unique apps for Android (11 collisions for Common and Popular sets). For iOS we collect 2,515 unique apps (60 collisions for Common and Popular sets). We see no collisions between the Random app set and other sets on either platform. Thus in total, we collect 5,079 unique apps, counting Android and iOS apps as different apps for the Common set.

# 4 METHODOLOGY

In this section, we detail the novel static and dynamic approaches we use to detect certificate pinning (RQ1) as well as to shed light on the implementation aspects related to it. Figure 1 presents an overview of our methodology.

## 4.1 Static Analysis

Static analysis involves studying apps without actually executing them. In this section, we discuss the parts of apps we study and the exact techniques used to infer whether certificate pinning is being implemented across apps.

*4.1.1 Configuration Files.* In Android, Network Security Configuration (NSC) files are used to customize network security settings without having to modify app code [20]. This technique allows apps to define general security settings or per-domain settings, with the option to specify certificates to trust, and to pin certificate hashes. We use static analysis to extract the *Android Manifest* file, which we parse to check if an app is using an NSC. If it is found, we extract the pertinent configuration file and parse that to obtain certificates and hashes that the app uses, extracting files as needed.

In iOS, App Transport Security Settings provide a similar feature of specifying pinned hashes in an app's configuration files [11]. We note that it is a recent feature , and is unavailable in the version of iOS used in our study. Because this feature was released close to our data crawls, we do not check for its prevalence in our datasets.

*4.1.2 Embedded Certificates.* Pinning implementations typically specify which certificates to pin in an app code. Therefore, we search for these certificates in app code by looking for any files ending with *.der, .pem, .crt, .cert,* and .cer extensions, or by extracting strings with delimiters such as "*-----BEGIN CERTIFICATE-----*". In addition, we also search for SHA-1/256 hashes of *SubjectPublicKeyInfo* (SPKI) field of certificates that is traditionally used in various protocols (e.g., HTTP Public Key Pinning [13] and DANE [12]) but also seen in some pinning implementations (e.g., Chrome [9] and Android OkHttp library [3]).

We decompile the Android apps using *Apktool*. As iOS apps are encrypted, we use Flexdecrypt[1] or Frida-iOS-Dump to extract decrypted payloads. We then employ fast recursive grep tool, *ripgrep*, to search for the regex patterns of interest (i.e., hashes[2] or certificates). In addition, we use *libradare2* to analyze strings from native libraries and/or executables present in the apps. We note that we do not attempt to de-obfuscate any decompiled files.

*4.1.3 Associated Certificates.* We search for certificates associated with *SubjectPublicKeyInfo* hashes found in apps using crt.sh certificate search [4] that indexes data from Certificate Transparency (CT) logs.

*4.1.4 Third-party Pinning Code.* Because we have information about the path in app code where each pin or certificate is found, we can use this information to shed light on the source of pinning code. To check for third-party pinning, we manually review all the certificate paths that appear in more than 5 apps, and infer whether the source is indeed a third party using publicly available knowledge (e.g., code in *sensibill* folder reflects their billing API in Android apps).

## 4.2 Dynamic Analysis

There are several reasons why support for pinning found statically in apps might not lead to pinning being used at runtime. For example, we may detect code that is unused (*e.g.,* due to a library that is never loaded, a library that provides optional support for pinning, or an outdated app version dynamically disabling pinning at run time). To address this, we use dynamic analysis; namely, we install and run apps on devices while collecting device logs and network traffic to determine if apps pin certificates at runtime. Thus, we consider results from dynamic analysis to be the ground truth about whether apps actually use pinning or not. In this section, we describe the components used in dynamic analysis in detail and how they tie together to detect pinning.

*4.2.1 Dynamic Pipeline.* We run apps from our datasets on real devices and collect network traffic. Our dynamic pipeline relies on automation frameworks for both platforms that control the devices via USB connections, and can install/run/uninstall apps on them. Our devices connect to a WiFi hotspot under our control. We use *mitmproxy* to proxy all the traffic from devices to servers, and to have the ability to MITM the traffic using an arbitrary CA certificate.

For Android tests, we use a Pixel 3 device running a factory system image of Android 11 (modified to include the *mitmproxy* certificate in the certificate store). For iOS tests, we use an iPhone X running iOS 13.6 (with trust for the *mitmproxy* certificate enabled).

[1] Flexdecrypt is faster than Frida-iOS-Dump because it does not require opening an app for decryption.

[2] We use the regular expression *sha(1|256)/[a-zA-Z0-9+/=]{28,64}*. The length allows us to search for hashes that are either base64- or hex-encoded.

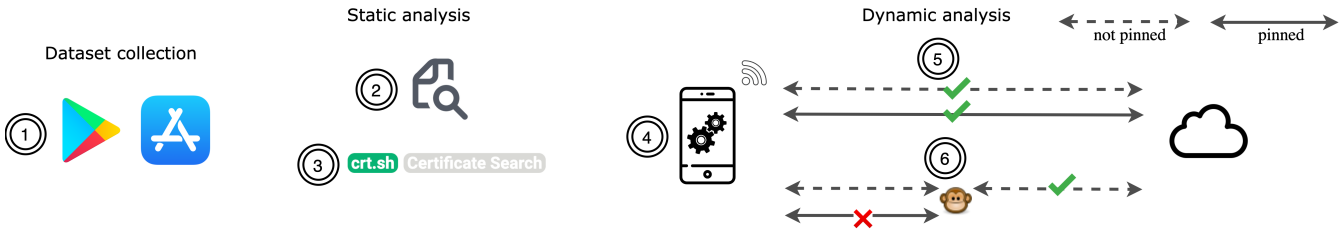



**Figure 1: Our methodology to detect certificate pinning. We (1) crawl Android and iOS apps, (2) search app contents for certificate files or hashes, (3) retrieve certificates corresponding to the hashes using publicly available Certificate Transparency logs, (4) launch every app on a real device and collect network traffic in two distinct settings: (5) when app traffic is not intercepted, and (6) when app traffic is intercepted through monkey-in-the-middle (MITM) technique. TLS connections that transmit data in the former setting but not the latter are identified as pinned.**

Our iPhone is jailbroken using Checkra1n to facilitate various aspects of the study (*e.g.,* app decryption for static analysis, pinning circumvention). Our choice to use the particular iPhone model and OS is based on the fact that there are no jailbreaks available for the latest combinations.

During dynamic testing, the automation framework installs one app at a time on the test device to ensure traffic isolation, waits 30 seconds to collect app traffic, and uninstalls the app before moving on to the next. For each dataset of apps, we run two experiments. Our first baseline experiment ("non-MITM" ) records TLS traffic triggered by apps without any interference. The second experiment ("MITM") runs with *mitmproxy* enabled, which tries to MITM any TLS connection. Based on the difference in app behavior in these two experiments, we extract information about pinned connections, as described in detail in the next section.

**App Interaction:** We experimented with various techniques to automate interacting with apps using UI Automator [16] for Android and a similar tool for iOS. While automation is itself feasible, the key issue is that apps on iOS and Android often present different UIs and we could not identify a general way to exercise the same functionality across platforms and thus could not conduct an apples-to-apples comparison. Given this, we also explored the potential for using *random* interactions that are commonly used in prior work (*e.g.,* [38], [36]). While the interactions would not be identical across platforms, they also should not have any particular bias overall. We conducted a small set of experiments where we used automated, random UI interactions, and we found no significant change in the number of domains contacted when compared to tests without any UI interactions. Thus, we do not perform any automated or manual interactions with apps in our study.

We varied the amount of sleep time (15 s, 30 s, and 60 s) from installing an app to uninstalling it, and heuristically found 30 seconds to be the best value for our study. More specifically, we found the average number of TLS handshakes performed by a small random sample of apps in the three settings to be 20.78, 23.5, and 24.62 respectively. As the vast majority of TLS connections are established within 30 seconds, we believe the diminishing returns beyond this point are not worth the additional wait.

*4.2.2 Detecting Pinned Connections.* A key challenge for detecting pinned connections is that it can be difficult to distinguish between a connection that fails due to TLS interception on a pinned connection, as opposed to a connection that fails for other reasons (*e.g.,* server-side issues). At a high level, our approach is to use a differential analysis, where we detect differences in connection behavior with and without TLS interception. Specifically, if a connection to a destination carries traffic beyond the handshake *without* TLS interception, and never carries traffic when *with* TLS interception, we mark the destination as pinned.

More specifically, we observed that a pinned TLS connection exhibits two key properties, and neither is unique to TLS interception. First, pinned TLS connections typically send failure signals via a TLS alert or TCP connection reset if an attempt is made to MITM them. However, these signals may also appear in an app traffic for reasons other than pinning (*e.g.,* TLS alert due to an unsupported protocol version). Second, pinning may result in a connection being successfully established, but it will never be used for transmitting useful application data if there is an attempt to MITM the connection. However, even without TLS interception, apps will create redundant connections and never use some of them to transmit application data. Thus, we need a way to account for such confounding factors. By comparing connection behavior in the two settings, we can attribute any observed connection failures to the presence of pinning.

We rely on the following definitions to evaluate a connection status:

**Used Connection:** To determine whether a TLS connection sends application data, we rely on the following tests. For TLS 1.2 or below, the presence of any "Encrypted Application Data" packets is sufficient to infer that the corresponding TLS connection is being used by a client. This inference does not work for TLS 1.3, where all encrypted records (data, alerts, or handshake messages) are disguised as TLS 1.2 "Encrypted Application Data" to reduce issues with middleboxes. Thus for TLS 1.3, we rely on the following two heuristics to identify connections that send application data: 1) clients either send more than two "Encrypted Application Data" packets, or 2) the second packet is not the same length as an encrypted TLS alert. The reasoning behind this is that the first encrypted client packet must always be "Client Handshake Finished" for successful connections according to the protocol specification, the second packet may or may not be an alert to indicate connection closure and third (if present) can only mean that application data has already been transmitted.

**Failed Connection:** We define a failed connection as any TLS connection that goes unused, *and* where the clients abort connections

with TCP RST or TCP FIN flags. This helps avoid false positives for cases where a connection simply remained unused in our experiments due to the limited recording time.

After collecting that status of connections, we evaluate which *destinations* are used at least once. Such information is readily available: 99% of the TLS traffic in our experiments have a non-empty SNI field, indicating the destination hostname for the connection. If a destination has any TLS connection that is used in the non-MITM setting, but TLS connections that always failed in the MITM setting, we mark it as pinned. We note that the heuristic to mark used connections does not need to be perfect, as we ultimately rely on whether an app behaves differently in the two experiment settings to determine pinning status.

### 4.3 Circumventing Pinning

Using the aforementioned methodology, we detect apps that implement pinning. In order to understand why apps implement pinning (RQ5), we attempt to look at the traffic sent in those pinned connections. To this end, we use Frida [7] to hook into various popular TLS libraries and disable certificate validation checks. When successful, this allows us to continue using our dynamic pipeline to MITM connections and obtain data that apps send to servers in pinned connections. We note that pinning circumvention is not guaranteed to succeed, as developers can always use custom TLS implementations rather than relying on popular ones. Using this approach, we were able to successfully circumvent pinning for ≈51.51% unique destinations on Android, and ≈66.15% unique destinations on iOS.

### 4.4 PII Analysis

Pinning can either be used to protect sensitive user data, or hide data collection from auditors. As we do not interact with the apps, we cannot check if pinned connections are being used to protect user data (*e.g.,* banking credentials). However, we can still check for the presence of other sensitive information that apps are known to collect from prior studies [27, 35, 37, 38].

More specifically, if we are successful in circumventing pinning, we inspect the decrypted application data to check for the presence of sensitive personally identifiable information (PII) that can harm user privacy. We also check whether PII presence differs significantly in the pinned vs non-pinned traffic. The PII we search for includes the following information for both platforms: *IMEI, advertisement ID, WiFi mac address, user email, state, city* and *latitude/longitude.* Although this list is not exhaustive by any means, it is sufficient for the purposes of this study as we are mainly interested in comparing PII prevalence across pinned vs non-pinned traffic, rather than finding out whether apps transmit any PII.

### 4.5 iOS Background Traffic

For Android, our manual analysis did not detect any background traffic that could interfere with our experiments. However, the situation for iOS turned out to be difficult to handle. First, we noticed TLS traffic to various Apple-controlled domains (icloud.com, apple.com and mzstatic.com) that spanned the whole duration of dynamic testing (mainly due to connection retries in MITM experiments). We simply excluded these destinations from our analysis.

Second, and more importantly, we also needed to ignore traffic to many first-party destinations for apps, because it might have been triggered by the OS, rather than the app. This is due to a feature in iOS that contacts all destinations that are marked as "associated" in an app's entitlements. When an app is installed, iOS triggers TLS communication with these destinations to verify that they are indeed controlled by the app's developer (by going to a specific pre-defined path). The purpose of this feature is to facilitate connections between the app and its website(s) (*e.g.,* to share credentials, navigate from browser to app). In our testing, we noticed that all this traffic appears as pinned, likely because the underlying iOS service does not trust our *MITM* certificate. Unfortunately, the traffic from OS exhibits a similar TLS fingerprint as regular app traffic. As such, we could not find a way to distinguish traffic to these destinations triggered by the app vs the iOS background service. To avoid falsely attributing pinning to apps, we ignored all associated destinations from an app's entitlements during our analysis. More specifically, 66% of apps did not specify any associated domain, so no traffic was excluded for these. For the rest, there were on average 4.8 unique associated domains present in the configuration. Note that this generic approach of excluding traffic can only cause false negatives (*i.e.,* filter out domains that actually pin), not false positives.

Since our goal is to conduct a head-to-head comparison of pinning prevalence in Android vs iOS, we re-ran our dynamic pinning detection pipeline for apps in the Common dataset that were found to be pinning in either Android or iOS through the prior methodology (72 apps in total). Our re-run was modified in the following way in order to avoid the issue with associated destinations: after installing an app, we waited 2 minutes to let the OS finish communicating with these first-party destinations. We launched the app afterwards, and then collected data for 30 seconds as we had done before. We use results from this re-run whenever we mention iOS common dataset in the rest of this paper. On a positive note, the limited re-run did not reveal any false-negatives in the initial run. Thus, we do not believe our methodology of handling iOS background traffic affects the results significantly.

## 5 RESULTS

We apply the techniques presented in Section 4 on each dataset we have collected in order to understand the prevalence of certificate pinning. We present the certificate pinning we find, per dataset and platform, for our static and dynamic analyses in Table 3. To help us compare our findings with prior studies, Table 2 summarizes pinning prevalence indicated in prior work. It is clear, however, that these prior studies entail a wide range of techniques for detecting pinning, use different app datasets, and were conducted over a wide time range. Thus, it is difficult to conduct a meaningful apples-to-apples comparison. Instead, to enable comparison with prior work, we focus on the NSC-based static-analysis technique used by multiple prior studies, using the datasets we collect.

**Pinning by Technique:** Prior research mainly relies on Network Security Configurations (NSCs) to detect pinning in Android [21, 33]. For our Android datasets, using the same approach, we find relatively few apps to pin (from 0.6% to 2.78% depending on the dataset). In contrast, our dynamic analysis technique finds up to 4

| Study | Year | Prevalence | Analysis | Dataset size | Dataset source |
|---|---|---|---|---|---|
| Fahl *et al.* [21] | 2012 | 10% | Dynamic | 20 | High-profile Android apps |
| Oltrogge *et al.* [29] | 2015 | 0.07% | Static | 639,283 | Apps from the Google Play store |
| Razaghpanah *et al.* [34] | 2017 | 2% | Dynamic | 7,258 | Android apps found in the wild |
| Stone *et al.* [40] | 2017 | 28% | Dynamic | 135 | Security sensitive Android apps |
| Possemato *et al.* [33] | 2020 | 0.62% | Static | 16,332 | Android apps using NSCs |
| Oltrogge *et al.* [30] | 2021 | 0.67% | Static | 99,212 | Android apps using NSCs |

**Table 2: Certificate pinning prevalence mentioned in prior work. Note that the variety of analysis techniques, datasets, and passage of time make direct comparisons difficult.**

| Dataset type | | Dynamic analysis | Static analysis | |
|---|---|---|---|---|
| | | | Embedded Certificates | Configuration Files* |
| Common (n = 575) | Android | 8.17% (47) | 26.96% (155) | 2.78% (16) |
| | iOS | 8.52% (49) | 22.96% (132) | - |
| Popular (n = 1000) | Android | 6.7% (67) | 19.7% (197) | 1.8% (18) |
| | iOS | 11.4% (114) | 33.4% (334) | - |
| Random (n = 1000) | Android | 0.9% (9) | 9.9% (99) | 0.6% (6) |
| | iOS | 2.5% (25) | 9.5% (95) | - |

**Table 3: Certificate pinning prevalence found using various methods across different datasets. Each cell denotes the number of applications with one instance of pinning, over the total number of applications in that dataset. (*) denotes the method used by prior work.**

times more pinning (*i.e.,* 1.8% to 6.7% in popular apps). Our findings suggest that apps likely have many options other than NSCs to deploy pinning.

We further find that pinning prevalence varies substantially for the novel static and dynamic approaches we develop. While static approaches provide us with potentially pinning apps, dynamic analysis gives us stronger evidence of pinning as we observe pinned behavior through network connections. Due to this reason, for the remainder of this paper, we call an app to be pinning if we find at least one instance of a pinned connection from the app in our dynamic analysis results.

**Pinning by Platform:** We find more pinning apps in iOS as compared to Android across all datasets. While we present a head-to-head comparison of apps in the next section, we find it interesting that even random iOS apps pin substantially more (2.5%) than the set of random Android apps (0.9%). Upon closer inspection, we notice that two destinations, `www.paypalobjects.com` and `firestore.googleapis.com`, get pinned in 10 and 5 apps respectively in the iOS random dataset. In comparison, for the Android random dataset, we do not find any common pinned destination. As such, increased pinning in iOS might be due to these and other third-party libraries that are pervasive in the iOS ecosystem, and choose to pin, as compared to the ones for Android.

**Pinning by Category:** To understand the characteristics of apps that use pinning (RQ2), we check whether apps belonging to certain categories pin more frequently than others. For each category from the two platforms, we normalize the number of apps that pin by the number of apps we have in that category. We present the top 10 app categories that pin in Android (Table 4) and iOS (Table 5).

We find that the top category in both platforms is "Finance," suggesting the use of pinning is to protect sensitive user data in these apps. The next two categories are "Social" and "Shopping," likely again due to the sensitivity of data shared on apps in these categories. In terms of categories within a platform, we find it interesting that none of the top 3 app categories for either platform appears in the respective top 10 pinned categories list. In fact,

| Category (Rank) | Pinning % | No. of Apps |
|---|---|---|
| Finance (9) | 22.99 % | 20 |
| Social (14) | 17.81 % | 13 |
| Events (28) | 15.0 % | 3 |
| Dating (33) | 14.29 % | 2 |
| Food & Drink (15) | 13.64 % | 9 |
| Shopping (18) | 12.96 % | 7 |
| Comics (32) | 12.5 % | 2 |
| Automobile (25) | 8.33 % | 2 |
| Travel (12) | 6.49 % | 5 |
| Weather (24) | 5.88 % | 2 |

**Table 4: Top 10 categories of apps that pin in Android across all datasets and pinning prevalence per category. Ranks indicate the popularity of a category in our dataset.**

| Category (Rank) | Pinning % | No. of Apps |
|---|---|---|
| Finance (9) | 20.63 % | 26 |
| Shopping (13) | 16.48 % | 15 |
| Travel (14) | 13.48 % | 12 |
| Social Networking (8) | 11.02 % | 14 |
| Photo & Video (6) | 10.67 % | 16 |
| Lifestyle (5) | 8.7 % | 14 |
| Food & Drink (11) | 8.49 % | 9 |
| Sports (16) | 8.16 % | 4 |
| Navigation (22) | 8.0 % | 2 |
| Books (19) | 7.69 % | 3 |

**Table 5: Top 10 categories of apps that pin in iOS across all datasets and pinning prevalence per category. Ranks indicate the popularity of a category in our dataset.**

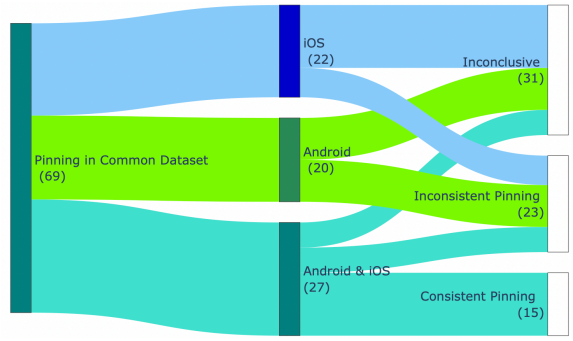



**Figure 2: Pinning found in the Common dataset split by platforms. We classify pinning found in this dataset into: Inconsistent, Consistent, and Inconclusive as defined in Section 5.1.**

"Games" is the most prevalent category across all our datasets, but does not appear in the top 10 pinned categories for either platforms.

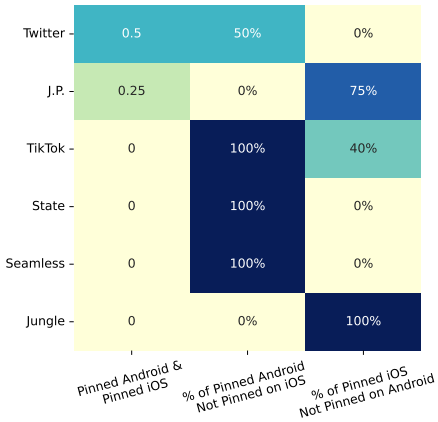
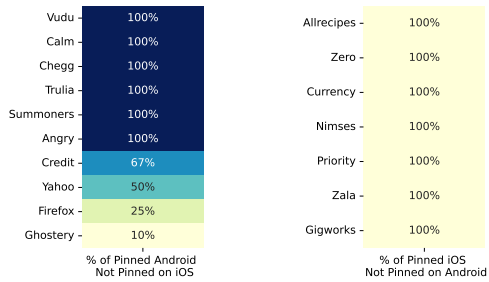



**Figure 3: Inconsistent pinning in apps that pin on both platforms. We see that the first two apps have overlapping pinned domains but are inconsistent as they pin domains on one platform and not on the other.**

## 5.1 Pinning in Common Apps

To understand whether pinning apps pin the same domains on both Android and iOS (RQ3), we consider apps from our Common dataset. From this dataset, we find 69 apps that pin on at least one platform. Of these, 27 apps pin on both Android and iOS, 20 apps pin solely on Android, and 22 apps pin solely on iOS. For each app, we compare the set of pinned and unpinned domains across platforms.

By definition, a single entity controls both Android and iOS versions of the same app in the common dataset, and one might hypothesize that they pin domains in the same way. However, we find in practice that they do not always do so, and thus define **inconsistent** and **consistent** pinning for this common dataset as




**(a) Android** **(b) iOS**

**Figure 4: Inconsistent pinning in apps that pin exclusively on one platform. Each cell represents the percentage of pinned domains on a platform found as not pinned on the other. For 6 Android and 7 iOS apps, all pinned domains appear as not pinned on iOS and Android respectively.**

follows. An app has *inconsistent pinning* if a domain pinned on one platform is not pinned on the other. An app has *consistent pinning* if it pins at least one common domain on both platforms and has no inconsistent pinning. Based on these definitions, we present Fig. 2. For a set of apps, we have inconclusive results, as domains pinned on one platform do not appear on the other at all. For these, we can not determine if the domains would be pinned or not, as we have not observed them.

**Apps Pinning on Both Platforms:** Of the 27 apps that pin on both platforms, we find that 15 apps have consistent pinning. For these apps, we aim to understand the number of common domains pinned on both platforms. To this end, we compare pinned domains on Android to pinned domains on iOS for each app. We find that 13 apps have the same set of domains pinned on both platforms. For the remaining two apps, we see that one domain is pinned on both platforms (Android pins one other and iOS pins two others).

To understand the inconsistent pinning apps, we compare pinned domain sets to not pinned domain sets. To compare similarities in pinned domains for two pinning sets, we use Jaccard indices. To compare a pinning set to a non-pinning set, we look at the percentage of pinning domains present in the non-pinning set. We use this instead of Jaccard indices here as we care about domains that are pinned in one set and not pinned in the other, as opposed to similarities between the two sets. We present a heatmap of these calculations in Fig. 3. Each inconsistent app is represented as a row with the first column giving us the overlap of pinned domains on both platforms. The second gives us the percentage of pinned domains on Android that appear as unpinned on iOS; the third gives the percentage of pinned domains on iOS that appear as unpinned on Android. Of the 6 apps, we see that 2 have overlaps of pinned domains; 3 have pinned domains on android that they do not pin on iOS and 3 pin domains on iOS that they do not pin on Android. For the remaining 6 apps, all values on such a heatmap would be 0. On analyzing these, we see that they share no common domains on the two platforms; thus all overlaps would be 0 and hence inconclusive.

**Apps Pinning on One Platform:** To understand how domains pinned on one platform are handled on another platform, we look at apps that pin exclusively on one platform. For this set, pinning consistencies are nonexistent as the other platform does not pin.

To understand these pinning inconsistencies, we compare pinned domains on one platform that appear as not pinned on the other. Apps that have pinned domains on one platform that do not appear as unpinned on the other are marked inconclusive. For 20 apps pinning exclusively on Android, we have 10 inconsistent and 10 inconclusive apps. For 22 apps pinning exclusively on iOS, we have 7 inconsistent and 15 inconclusive apps.

We calculate the percentage of pinned domains that appear as not pinned on the other platform and plot a heatmap of these in Figure 4. We see that 7 Android apps have all traffic pinned on Android appearing as not pinned on iOS. All apps on iOS marked as inconsistent (7) have all pinned domains appearing as not pinned on Android. Thus, for both these sets of apps, we see that developers talk to common domains and pin them on one platform while not pinning them on the other. This indicates that the pinning policies of these apps is inconsistent and vary greatly based on the platform.

## 5.2 Pinning in Popular vs Random Apps

To understand the prevalence of pinning in popular and arbitrary apps (RQ2), we apply our detection methodology on the popular 1k and random 1k datasets. For Android, we find that 67 apps from popular and 9 apps from random datasets use pinning. For iOS, we find 114 apps from popular and 25 apps from random datasets use pinning. Thus, we see that pinning is more prevalent on iOS as compared to Android.

To further understand the nature of pinning, specifically the parties involved in pinning in an app, we dig deeper into the number of pinned connections and present the results in Figure 5. Each bar on the x-axis represents an app that pins at least one domain, split by dataset and platform. The y-axis shows the percentage of pinned and not pinned domains that each app contacts in our tests. Blue represents pinned domains and green represents not pinned domains. We divide domains contacted by an app into first and third party, attributing each domain for an app using various points of information (whois data, certificate subject names, *etc.*). We annotate each bar with first and third party data marking first parties with dark and third parties with light colors.

We see that almost all Android apps that contact first party domains also pin those domains (28); with a single exception, Trulia (`com.trulia.android`). On the other hand, Android apps that pin third parties (51) rarely pin all third parties (4); many apps pin some third parties and do not pin others (47).

In contrast, for iOS we observe many cases where first parties are not pinned (18), often when other first parties are pinned (6, dark blue and dark green on the same bar). Similar to Android, many iOS apps pin all first party domains they contact (39). All iOS apps in our dataset that pin third parties contact other not pinned third parties (99).

Based on the results in Figure 5, we observe that apps on both platforms almost always have inconsistent pinning practices; domains (regardless of first or third party) are selectively pinned, disregarding other traffic. Only 5 apps on Android (AskURA, Auto Kiosk, Edmtrain, FFBaD, and Private Fostering Awareness) and 4 apps on iOS (Bank of America, CandyCrush, Facebook, and Surge proxy) pin all domains they contact.

| Platform | Default PKI | Custom PKI | Data Unavailable |
|---|---|---|---|
| Android | 163 | 4 | 11 |
| iOS | 238 | 1 | 14 |

**Table 6: Type of PKI used by pinned destinations. The majority of pinning happens with certificates that tie to the default PKI.**

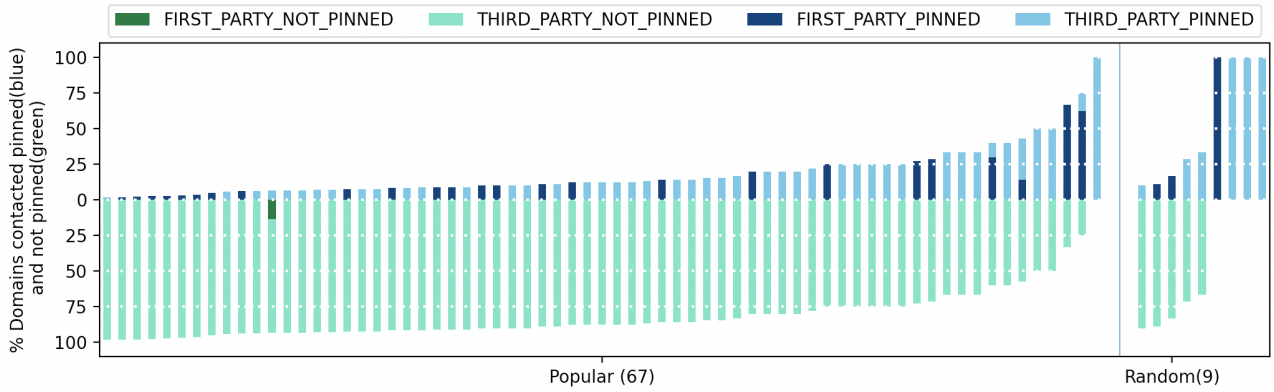
## 5.3 Certificate Analysis

In this section, we explore how certificate pinning gets implemented in apps from the two platforms (RQ4). To do this analysis, we gather certificate chains that are served at destinations found to be pinning via dynamic analysis, as well as, the certificates found in apps using static analysis. We remind the reader that our static analysis techniques (i) search for raw certificates present in an application, and (ii) attempt to fetch certificates associated with any SPKI hashes present in the application. More specifically, our static analysis techniques discovered 966 unique certificates across all apps present in raw format, as well as 170 unique certificates associated with 50% of the unique pins from all apps. Using this static and dynamic certificate data, we shed light on the following aspects of pinning implementations:

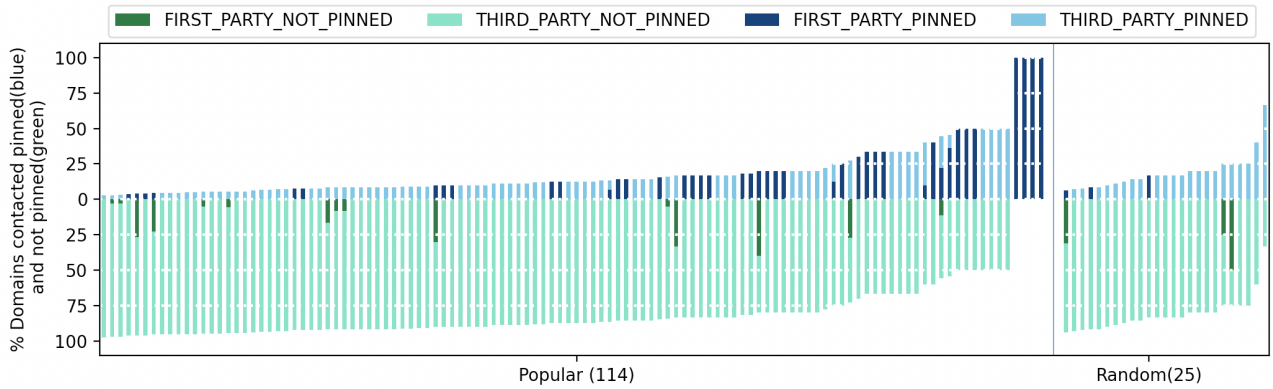
*5.3.1 The Public Key Infrastructure (PKI) Used.* Both Android and iOS come pre-bundled with a default set of root CA certificates (the "default PKI"). We note that in the case of Android, Original Equipment Manufacturers (OEMs) may add additional root certificates in addition to those included in Android's Open Source Project (AOSP) [22, 42]. Apps that wish to implement pinning can either pin certificates that still tie to the default PKI, or use a "custom PKI" altogether by trusting their own root CA. To understand which of these two mechanisms are prevalent in apps, we validate certificate chains served at all pinned destinations using OpenSSL, configured with the latest version of Mozilla CA certificate store [5]. Further, we manually review the ones OpenSSL could not validate to confirm that they are indeed certificates tied to custom PKIs. Our results, summarized in Table 6, reveal that the vast majority of pinning happens with default PKI in use.

Interestingly, we also find two cases, one per platform, where the destination presents a self-signed certificate, rather than a chain. Although these destinations are reaping the benefits of certificate pinning, they are likely missing the flexibility provided by a PKI. To illustrate this, we note that the expiry dates for these certificates are 27 and 10 years. Due to this long validity period, and because these certificates cannot be revoked (revocation only applies to leaf certificates), any key compromise will mandate app updates to protect connection security.

*5.3.2 Pinning Root vs Leaf Certificate.* Apps can choose to either pin the root or a leaf certificate from the certificate chain, with the former offering more flexibility while the latter offering more security. More specifically, pinning a leaf certificate protects the TLS connection from all CAs, including the issuer. But on the other hand, leaf certificates have shorter expiry periods, and their keys are more likely to be rotated for security reasons. As such, pinning leaf certificates demands more management, and can even render applications unusable if the apps are not updated to reflect the latest leaf served from a destination.




**(a) Android Pinning**




**(b) iOS Pinning**

**Figure 5: Percentage of domains contacted that are pinned vs not pinned. Dark colors represent first parties and light colors represent third parties. Each bar represents an app, from the popular 1k and random 1k datasets for Android and iOS respectively.**

To understand which type of certificates apps choose to pin, we cannot only rely on dynamic data alone as this data reveals certificate chains presented at pinned destinations. Rather, we need to investigate which certificate in the chain is likely pinned in an app's code using static analysis methods (section 4.1.2). In our analysis, we find ≈31% of pinning apps across the two platforms, for which there is at least one certificate that appears in both static and dynamic data (certificate matching is done in terms of the Common Name). We find the majority of these certificates to be CAs (80/110), and the remaining (30/110) to be leaf certificates.

*5.3.3 Pinning Entire Certificate vs Its Key.* As mentioned earlier, pinning leaf certificates can lead to unavailability issues if the certificates are updated at the server but an outdated app version is used, or if the developers forget to update pinned data on the app. There is one exception since pinning can be done via a certificate's Subject Public Key Information (SPKI): app developers can update certificates on the servers as long as the certificate key remains unchanged. Our data indicates that this is indeed how app developers implement pinning. More specifically, out of the 30 leaf certificates that we found to be pinned in the previous section, 24 of them were pinned via SPKI hashes. The remaining 6 leaf certificates are present in their raw format in the apps, thus the developer could either pin the whole certificate or just the public key. In 5 of these 6 cases, we notice that destinations serve new leaf certificates during dynamic testing, which still result in pinned connections. This suggests that app developers likely pinned public keys for these certificates. Although this is good news for app usability, it also implies that certificate keys are reused which, in-turn, defeats the purpose of certificate renewals.

*5.3.4 Subverting Proper Certificate Validation.* Because certificate pinning only protects TLS connections against particular attacks, any pinning implementation still needs to conduct other certificate validation checks as defined in the TLS protocol (e.g., certificate subject name match, date validation) to protect against other attacks [40]. To see if any apps that use pinning bypass other standard certificate validation checks, we check for expiry dates of certificates served at pinned destinations. We do not find any certificates that are expired but were considered valid by apps during dynamic analysis. As such, we do not find any evidence of apps subverting normal certificate validation to only rely on pinning as the protection mechanism.

*5.3.5 Third-party Frameworks That Introduce Certificates.* We finally look at the package code paths in apps where our static analysis detects certificates and/or pins to attribute the behavior to first-party or third-party code. We observe that many of these paths appear in multiple apps. Upon manually investigating for the top

| Platform | Framework | # apps |
|---|---|---|
| Android | Twitter | 29 |
| | Braintree | 27 |
| | Paypal | 25 |
| | Perimeterx | 9 |
| | MParticle | 9 |
| iOS | Amplitude | 45 |
| | Stripe | 34 |
| | Weibo | 24 |
| | FraudForce | 16 |
| | Adobe Creative Cloud | 13 |

**Table 7: Top 5 third-party frameworks that include certificate in Android and iOS. We combine paths where certificates are found across apps and provide occurrences here.**

common paths with certificates, and removing generic ones (such as config.json), we present the list of various third-party frameworks that we identify to likely be introducing certificate pinning logic to the apps (Table 7). For some of these, we are able to trace the pinning code in their open-source repositories (e.g., Twitter SDK, MParticle SDK). We note that some of these frameworks are also associated with popular pinned domains from our dynamic analysis (*e.g.,* config2.mparticle.com, *.perimeterx.net). Last, we believe that the end-points that did not appear during our dynamic analysis are likely the ones for which our monkey was unable to automatically trigger the associated code paths. We particularly believe this to be the case with Paypal that, appears as a popular pinned domain in iOS, but never appears in Android (except for the Paypal app). Overall, our analysis reveals social networks, payment processing systems, and app analytics frameworks are the common sources of third-party code that introduces certificate pinning in apps.

## 5.4 Connection Security

In this section, we explore whether apps that use pinning also adopt other security practices in their pinned TLS connections (RQ5). More specifically, we check whether these TLS connections advertise support for bad ciphersuites (*e.g.,* DES, 3DES, RC4 or EXPORT) that are susceptible to many attacks. We compare their prevalence with connections from all apps to contrast security practices of apps that implement pinning. Our results are presented in Table 8. The "Overall" column shows the percent of all apps in a dataset that have at least one TLS connection with bad ciphers, while the "Pinning apps" column shows the percent of apps with certificate pinning that have at least one *pinned* TLS connection with bad ciphers.

Across all three iOS datasets, we see an increase in connection security of pinning connections when compared to the overall connections in every set. Weak ciphers drop from: 93.39% to 55.77% for Common iOS, 95.2% to 46.09% for Popular iOS, and 82.6% to 52.94% for Random iOS datasets. However, trends in Android are more nuanced. For the Common Android dataset, we see that pinning apps reduce connection security as the percentage bad ciphers in pinning apps is higher (23.4%) than that of the overall dataset (8.35%). But for the Popular and Random Android datasets, we see an increase in connection security of pinning connections as compared to other

| Dataset | | Bad Ciphers | |
|---|---|---|---|
| | | Overall | Pinning apps |
| Common | *Android* | *8.35%* | *23.4%* |
| | iOS | 93.39% | 55.77% |
| Popular | Android | 18.3% | 1.49% |
| | iOS | 95.2% | 46.09% |
| Random | Android | 3.1% | 0.0% |
| | iOS | 82.6% | 52.94% |

**Table 8: Weak ciphers found in pinned vs all connections across all datasets for Android and iOS. In general, we see pinning apps increase connection security in pinned connections as they disable weak ciphers more often than other apps in the dataset. The Common Android dataset (italics) is an exception to this trend supporting weak ciphers more often than the rest of the dataset.**

| Platform | PII | Pinned | Non-Pinned |
|---|---|---|---|
| iOS | Ad. ID* | 25.85 % | 18.06 % |
| | City | 0 % | 0.94 % |
| | State | 0 % | 0.31 % |
| | Lat./Lon. | 0 % | 0.04 % |
| Android | Ad. ID | 25.74 % | 19.96 % |
| | Email | 0.99 % | 0.52 % |
| | State | 0.99 % | 1.12 % |
| | City | 0 % | 0.45 % |

**Table 9: PII found in pinned connections, and how the prevalence differs from non-pinned TLS connections. (*) marks results that are statistically significant.**

apps in those sets. Weak ciphers drop from 18.3% to 1.49% for the Popular set and from 3.1% to 0.0% for the Random set.

Thus, with the exception of the Common Android dataset, our data suggests that pinning apps likely have better connection security for their pinned connections when compared to non-pinning apps on both Android and iOS.

## 5.5 PII in Pinned vs Non-Pinned Traffic

Since pinned TLS connections are harder to inspect by device users and auditors, in this section we try to understand whether app developers implement pinning in order to hide sensitive PII data collection, rather than to improve user security (RQ5). To do so, we inspect PII prevalence in decrypted TLS connections for all apps that implement pinning using the methodology described in 4.4.

Our results are presented in Table 9 and reveal what PII is found in pinned traffic, and how does the prevalence differs for non-pinned traffic. Since the number of non-pinned destinations is orders of magnitude more than pinned ones on both platforms, we cannot simply compare the PII prevalence across the two categories. As such, we highlight the results where differences in PII prevalence are statistically significant (found using Chi-square test of independence with a p-value < 0.05). We find that advertisement ID is the key identifier that appears substantially in both pinned and non-pinned traffic. Although it appears more in pinned traffic, the differences we see are statistically significant in only one platform. We do not find substantial presence of other identifiers that we

checked for. As such, our results suggest that app developers likely do not use pinning as a method to hide PII data collection.

## 5.6 Limitations

We discuss limitations of our methodology here. We also claim to find a lower bound of certificate pinning, which remains unaffected by these limitations.

**Embedded Certificates:** We search for certificates embedded in apps, but could miss certificate for various reasons, such as apps using obfuscated code, reconstructing certificates at run-time, storing certificates in non-standard formats, *etc.*

**Partial Observation:** Our dynamic testing is limited; we do not explore all code paths of an app. Thus, we miss certificate pinning that is not triggered during our testing. Similarly, we do not have ground-truth knowledge about all of the PII that apps collect. Our analysis instead is limited a subset of PII that we could infer automatically in network traffic.

**iOS Background Traffic:** As discussed in Section 4.5, we exclude iOS "associated" domains from our pinning calculations to avoid introducing noise due to OS-initiated background traffic. This may lead to an underestimation of pinning on iOS.

**Limited App Interaction:** Though we explored automated interactions with apps, we found they had a limited impact on results. We did not log into or interact with apps after doing so. Thus, we potentially miss pinned connections that would appear in such scenarios.

**Dataset:** Given that the dataset is collected from official stores, we do not capture the prevalence of pinning outside of official channels. Similarly, we do not cover prevalence for paid apps. Lastly, we tested a relatively modest number of apps (≈3,000), largely due to scalability constraints for dynamic testing. While we partially mitigate this by selecting different collections of apps (popular, random), our study nonetheless represents only a sample of all available apps.

## 5.7 Discussion and Future Work

**Pinning Inconsistencies:** We introduce the concept of *inconsistent* and *consistent* pinning for the common dataset of apps. Apps from the common dataset are developed and maintained by the same entity (developer, company, *etc.*). Thus, we expect the pinning policies to be consistent across these two mobile platforms. We find that this is rarely the case, with less than half the apps having completely consistent pinning. This indicates that the security practices of the same entity are different on Android and iOS, and is an interesting finding as it is unexpected.

We argue that pinning consistently, across platforms, is good practice. Although codebases for various platforms might vary, the reasoning behind pinning should be the same. We can only speculate about the reasons for such differences, *e.g.,* they could be due to different pinning APIs across OSes causing confusion/inconsistency (*e.g.,* as found by Oltrogge *et al.* [29]), or due to developers using different threat models for iOS compared to Android.

**Developer Survey:** This work revealed practices that cannot be explained by our dataset alone. By surveying developers who use pinning, we can better understand the reasoning behind pinning, including why there are inconsistencies between Android and iOS. Such a survey can also help the community to better understand deployment/maintenance requirements for pinning, and compile a better set of guidelines for developers that wish to use certificate pinning.

**App Exploration:** An orthogonal problem we encountered during our study was app exploration. We tested random automated interactions with apps but found no significant change in traffic generated by the apps with or without these interactions. Developing a tool that automatically interacts with apps (signing up, logging in *etc.*) would be useful for various future studies.

**Pinning Circumvention:** In this paper we used existing techniques to circumvent pinning to study data that is protected behind pinned connections. The number of connections we circumvented was limited (≈50% destinations). We leave it to future works to develop techniques that can circumvent a larger number of pinned connections, enabling studies of data protected by pinning.

# 6 CONCLUSION

This paper conducted the first large-scale study of certificate pinning across both Android and iOS apps. We found significantly higher prevalence of pinning than in prior studies, with at least 11% of popular iOS apps and 6.7% of popular Android apps doing so. Interestingly, we found that pinning behavior varies significantly across platforms, even for the same app. Based on our analysis, pinning is commonly added by third-party libraries and is likely deployed for the protection of financial data, with little evidence that pinning is used primarily to protect (non-credential) personal data. In future work, we will explore how results change with more app interactions, both automated and manual.

# 7 ETHICS

This paper does not entail human subject research. All tests were conducted using accounts set up for the sole purpose of our testing.

Our methodology requires crawling app stores, and we used low crawling rates with accounts that are easily identified as being used for research purposes (in case the platforms took notice of our crawls and needed to contact us). Similarly, while crawling the AlternativeTo website, we limited our crawler to request 1 page/second and included our contact information in the User-Agent field. We received no complaints about our crawls, nor were any of our accounts disabled or rate limited in any way.

# ACKNOWLEDGEMENTS

The authors would like to acknowledge the anonymous reviewers for their constructive feedback. This project is funded by the NSF (CNS-1564329, CNS-2053363, SaTC-1955227), the EU's H2020 Program (TRUST aWARE Project, Grant Agreement No. 101021377) and the Spanish Ministry of Science (ODIO Project, PID2019-111429RB-C22). Dr. Narseo Vallina-Rodriguez is also funded by a Ramon y Cajal Fellowship from the Spanish Ministry of Science & Innovation. This research has received funding from the Vienna Science and Technology Fund (WWTF) through project ICT19-056, as well as SBA Research (SBA-K1), a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

# REFERENCES


[1] [n. d.]. AlternativeTo - Crowdsourced software recommendations. https://alternativeto.net/. ([n. d.]). (Accessed on 05/17/2022).

[2] [n. d.]. App Store Downloads on iTunes. https://apps.apple.com/us/genre/ios/id36. ([n. d.]). (Accessed on 05/11/2022).

[3] [n. d.]. CertificatePinner (OkHttp 3.14.0 API). https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html. ([n. d.]). (Accessed on 01/17/2021).

[4] [n. d.]. crt.sh | Certificate Search. https://crt.sh/. ([n. d.]). (Accessed on 05/17/2022).

[5] [n. d.]. curl - Extract CA Certs from Mozilla. https://curl.se/docs/caextract.html. ([n. d.]). (Accessed on 05/19/2022).

[6] [n. d.]. Dell does a Lenovo: ships laptops with rogue root CA - gHacks Tech News. https://www.ghacks.net/2015/11/23/dell-does-a-lenovo-ships-laptops-with-rogue-root-ca/. ([n. d.]). (Accessed on 05/17/2022).

[7] [n. d.]. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. ([n. d.]). (Accessed on 11/15/2021).

[8] [n. d.]. google-play-scraper · PyPI. https://pypi.org/project/google-play-scraper/. ([n. d.]). (Accessed on 05/18/2022).

[9] [n. d.]. ImperialViolet - Public key pinning. https://www.imperialviolet.org/2011/05/04/pinning.html. ([n. d.]). (Accessed on 01/17/2021).

[10] [n. d.]. matlink/gplaycli: Google Play Downloader via Command line. https://github.com/matlink/gplaycli. ([n. d.]). (Accessed on 05/18/2022).

[11] [n. d.]. NSPinnedDomains | Apple Developer Documentation. https://developer.apple.com/documentation/bundleresources/information_property_list/nsapptransportsecurity/nspinneddomains. ([n. d.]). (Accessed on 05/19/2022).

[12] [n. d.]. RFC 6698 - The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. https://tools.ietf.org/html/rfc6698. ([n. d.]). (Accessed on 01/17/2021).

[13] [n. d.]. RFC 7469 - Public Key Pinning Extension for HTTP. https://tools.ietf.org/html/rfc7469. ([n. d.]). (Accessed on 01/17/2021).

[14] [n. d.]. Rogue web certificate could have been used to attack Iran dissidents | Google | The Guardian. https://www.theguardian.com/technology/2011/aug/30/faked-web-certificate-iran-dissidents. ([n. d.]). (Accessed on 05/17/2022).

[15] Android Developer. 2021. Security with HTTPS and SSL (version updated 2021-01-26). https://web.archive.org/web/20210301223141/https://developer.android.com/training/articles/security-ssl. (January 2021). (Accessed on 05/18/2022).

[16] Android Developer. 2022. Write automated tests with UI Automator. https://web.archive.org/web/20220907074832/https://developer.android.com/training/testing/other-components/ui-automator. (March 2022). (Accessed on 09/07/2022).

[17] Apple Developer. 2021. Identity Pinning: How to configure server certificates for your app. https://developer.apple.com/news/?id=g9ejcf8y. (January 2021). (Accessed on 05/18/2022).

[18] Can I Use. 2022. HTTP Public Key Pinning. https://caniuse.com/?search=hpkp. (2022). (Accessed on 05/18/2022).

[19] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[20] Google Developer. [n. d.]. Network Security Configuration. https://developer.android.com/training/articles/security-config. ([n. d.]). (Accessed on 09/08/2021).

[21] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[22] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An analysis of pre-installed android software. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1039–1055.

[23] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. 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*. 38–49.

[24] Google Developer. 2012. Jelly Bean. https://developer.android.com/about/versions/jelly-bean.html#android-4.2. (2012). (Accessed on 05/18/2022).

[25] Google Developer. 2021. Security with HTTPS and SSL. https://developer.android.com/training/articles/security-ssl. (October 2021). (Accessed on 05/18/2022).

[26] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert H. Deng. 2013. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[27] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. 2022. Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.

[28] Zane Ma, James Austgen, Joshua Mason, Zakir Durumeric, and Michael Bailey. 2021. Tracing your roots: exploring the TLS trust anchor ecosystem. In *Proceedings of the 21st ACM Internet Measurement Conference*. 179–194.

[29] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. 2015. To Pin or Not to {Pin—Helping} App Developers Bullet Proof Their {TLS} Connections. In *24th USENIX Security Symposium (USENIX Security 15)*. 239–254.

[30] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. 2021. Why Eve and Mallory Still Love Android: Revisiting TLS (In) Security in Android Applications. In *30th USENIX Security Symposium (USENIX Security 21)*.

[31] Open Web Application Security Project (OWASP). 2021. Certificate and Public Key Pinning. https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning. (December 2021). (Accessed on 05/18/2022).

[32] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. 2016. CRiOS: Toward large-scale iOS application analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 33–42.

[33] Andrea Possemato and Yanick Fratantonio. 2020. Towards HTTPS Everywhere on Android: We Are Not There Yet. In *29th USENIX Security Symposium (USENIX Security 20)*. 343–360.

[34] 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 the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 350–362. https://doi.org/10.1145/3143361.3143400

[35] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, Phillipa Gill, et al. 2018. Apps, Trackers, Privacy, and Regulators A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[36] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*. 603–620.

[37] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. Bug Fixes, Improvements, ... and Privacy Leaks – A Longitudinal Study of PII Leaks Across Android App Versions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[38] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)*.

[39] Statista. 2022. Number of mobile app downloads worldwide from 2016 to 2021. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/. (2022). (Accessed on 05/18/2022).

[40] Chris McMahon Stone, Tom Chothia, and Flavio D Garcia. 2017. Spinner: Semi-automatic detection of pinning without hostname verification. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 176–188.

[41] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. 2020. iOS, Your OS, Everybody's OS: Vetting and Analyzing Network Services of iOS Applications. In *29th USENIX Security Symposium (USENIX Security 20)*. 2415–2432.

[42] Narseo Vallina-Rodriguez, Johanna Amann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. 2014. A tangled mass: The android root certificate stores. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 141–148.

# A IOS DATASET COLLECTION

The key challenge is that iOS apps are encrypted with keys hard-coded in Apple devices, as well as the user key associated with an app download. In addition, there are no public APIs that can be used to fetch app contents. While Tang et al. [41] collected a large corpus of iOS apps using various novel techniques, they unfortunately did not open-source their download tool, and we were unable to reproduce it due to lack of some key details. Correspondence with the authors also did not reveal these important details.

Our approach was thus inspired by earlier work that automates GUI interaction with (the now deprecated) iTunes 12.6 application [32], and was also followed by concurrent work on PII leakage in Android compared to iOS apps [27]. The process is semi-automated, as we occasionally needed to manually fix various issues (*i.e.,* re-authenticate); the inability to download apps in a fully unattended way is the main reason we restricted the scale of our analysis to thousands of iOS apps. To be consistent, we chose the same number of apps from Android.