Racing for TLS Certificate Validation: A Hijacker's Guide to the Android TLS Galaxy

Sajjad Pourali†[1], Xiufen Yu†[1], Lianying Zhao[2], Mohammad Mannan[1], and Amr Youssef[1]

[1]Concordia University, Montreal, Canada
[2]Carleton University, Ottawa, Canada

Abstract

Besides developers' code, current Android apps usually integrate code from third-party libraries, all of which may include code for TLS validation. We analyze well-known improper TLS certificate validation issues in popular Android apps, and attribute the validation issues to the offending code/-party in a fine-grained manner, unlike existing work labeling an entire app for validation failures. Surprisingly, we discovered a widely used practice of overriding the global default validation functions with improper validation logic, or simply performing no validation at all, affecting the entire app's TLS connections, which we call *validation hijacking*. We design and implement an automated dynamic analysis tool called *Marvin* to identify TLS validation failures, including validation hijacking, and the responsible parties behind such insecure practices. Among 7826 apps from a Chinese app store and Google Play analyzed by Marvin, we found many occurrences of insecure TLS certificate validation instances (55.3% of the Chinese apps and 6.4% of the Google Play apps). Validation hijacking happens in 34.3% of the insecure apps from the Chinese app store and 21.1% of insecure Google Play apps. A network attacker can exploit these insecure connections in various ways, e.g., to compromise PII, app login and SSO credentials, to launch phishing and other content modification attacks, including code injection. We observed that most of these vulnerabilities are related to third-party libraries used by the apps, not the app code created by app developers. The technical root cause enabling validation hijacking appears to be the specific modifications made by Google in the OkHttp library integrated with the Android OS, which is used by many developers by default, without being aware of its potential risks. Overall, our findings provide valuable insights into the responsible parties for TLS validation issues in Android, including the validation hijacking problem.

1 Introduction

According to GlobalStat [47], Android smartphones have exceeded 72% of the smartphone market share in 2023, indicating the increasing importance of Android security. Despite the success and prevalence of TLS for app security, its use is marked with many security issues (see e.g., [29, 35, 36, 50]). Existing studies are exemplified by Wang et al. [50, 51] identifying vulnerable TLS implementations with code snippets; and Oltrogge et al. [35] demonstrating that Google's Network Security Configuration (NSC) and Google Play Safeguards failed to detect vulnerable TLS implementations.

Among the various ways TLS can go wrong, we focus on the validation of the TLS certificate, which is the foundation of establishing secure communication. Existing research largely considers validation issues per-app, even though validation functions from several entities, including the app developer and authors of various libraries (e.g., ads, analytics) used in a given app, may be involved in an app's TLS connections. Beyond finding the responsible parties behind TLS validation problems, perhaps more importantly, existing work also does not explore how or whether these parties can affect each other's validation effort without the other parties' knowledge. Note that stealthy data collection by third-parties is a well-studied problem in the literature (e.g., [24, 26, 42, 44, 49]).

In this paper, we detail the design and implementation of an analysis tool called *Marvin*[1] to attribute TLS certificate validation problems to more specific parties, e.g., the app developer or various libraries included in the app, i.e., not just the app as a whole. When validation mistakes happen in a library affecting other entities in the app (e.g., the app developer and other libraries), the impact can be significantly higher (compared to an app developer's faulty TLS validator) as a library may be used by a large number of apps. Using Marvin, we systematically analyze insecure TLS connections in Android apps resulted from improper certificate validation

† Equal contribution.

[1] Marvin is the paranoid but super-intelligent Android in The Hitchhiker's Guide to the Galaxy series: https://en.wikipedia.org/wiki/Marvin_the_Paranoid_Android.

by: 1) exercising the states of the apps with UI interaction to maximize the coverage of connections; 2) tracing real-time code execution for TLS validation-related functions; and 3) capturing corresponding network traffic with a man-in-the-middle proxy (mitmproxy [15]). We use certificates with common and well-documented validation issues in our proxy, so that the statistics from the measurements and the attribution results are self-evident and built on verified grounds.

For each established connection through the proxy, i.e., when the invalid certificate is accepted by the app, Marvin extracts various metadata for attribution purposes, e.g., destination address, app name, functions involved in the validation. Our fine-grained attribution process led us to discover a highly insecure practice we term as *validation hijacking*: the app's or an included library's code globally setting default insecure TLS validation functions, resulting in a significant number of certificates being validated by the set function (simply bypassing the validation in many cases), unbeknownst to the code initiating the connections (after such global modifications), e.g., the app developer's code or a third-party library's code.

We use Marvin to analyze a set of representative apps (in terms of user base) collected from Google Play and 360 Mobile Assistant (Qihoo 360, a popular Chinese app store [18], which allows automated app download). Marvin identifies many instances of validation issues, a majority of which are in Chinese apps due to their inclusion of a few very popular but problematic SDKs, e.g., Tencent Bugly [17]. We traced back the root cause of validation hijacking to specific modifications that honor the improper override (introduced by Google) to the OkHttp library included in the Android OS, although other HTTP libraries can also cause similar issues.

Our seemingly straightforward approach is faced with several challenges. For instance, numerous apps (especially those from the Chinese app store) are protected with strong state-of-the-art commercial packers, mainly against dynamic analysis as well as code obfuscation, rendering execution tracing very difficult. Furthermore, our attribution process requires tracing Java interfaces, which cannot be directly hooked; instead, their implementations need to be hooked, necessitating an exploration of the entire code loaded into memory to identify their implementation and utilize Java reflection to match them with their definitions for the hooking process. Some apps also do not adhere to the system proxy settings, and network traffic encompasses a mix of connections originating from the entire Android platform, system apps, and the target app, making it difficult to capture and filter traffic from the target app only.

We partially address such challenges so as to be able to conduct our analysis, and ensure that our findings are correct and the reported numbers are a lower bound (i.e., we may have false negatives, but no false positives). For example, we make use of eBPF [9], a kernel-level restricted but privileged execution environment, to avoid being detected, to separate a target app's network traffic, and to redirect connections not respecting the system proxy setting.

Contributions and notable findings.

1. We design and implement *Marvin*, an automated tool to perform a large-scale automated analysis of common TLS certificate validation issues, and to identify the responsible code/party for such validation failures, including a novel but very damaging case we call *validation hijacking*. We will open source Marvin.

2. We report on the prevalence of TLS validation failures across 7826 Android apps from two different app stores, consisting of 2765 Chinese ecosystem apps and 5061 Google Play apps—all automatically analyzed by Marvin. We identified that 55.3% of the apps in the Chinese app store and 6.4% of the Google Play apps have at least one connection with its TLS certificate not properly validated, which can lead to MITM attacks.

3. We conduct a fine-grained attribution, which, for the first time, attributes the TLS certificate validation failures to either an app's code, or the code of one of the multiple libraries an app uses. Such attribution should make a significant difference in remediation (vs. simply criticizing insecure developer practices). No past studies identified the truly responsible party behind such failures—i.e., the entity performing faulty validation, which indeed can be different than the originator of the TLS connection.

4. We draw attention to a pervasive but never discussed phenomenon, TLS certificate *validation hijacking*, especially, given many instances of such hijacking as detected by Marvin: 524/1529 (34.3%) of the Chinese apps, and 68/322 (21.1%) of the Google Play apps with TLS validation issues suffer from validation hijacking.

5. We identify the high-profile libraries which perform validation hijacking by overriding `setDefaultSSLSocketFactory()` or `setDefaultHostnameVerifier()`, e.g., Tencent Bugly, Baidu Location SDK, and Bytedance SDK. However, as we uncover, such hijacking is ultimately enabled by Google's modifications to the OkHttp library (the default Android HTTP client).

6. We show concrete issues resulting from the identified TLS validation issues, both in terms of information disclosure and content modification attacks by an on-path network adversary. We found that 88.8% of the Chinese apps and 37.6% of the Google Play apps with TLS validation issues use insecure TLS channels to transmit sensitive information. Our case studies with selected apps highlight the severity of the identified risks (tested on our own accounts): user credentials including one-time passwords and SSO login credentials (e.g., Google, Facebook, Twitter) compromise allows account takeover, in-app phishing, and even remote code execution.

We will open-source our tool on: https://github.com/Madiba-Research/Marvin.

Disclosure. In the process of disclosing our findings, we reached out to Google and the libraries that hijack the TLS verifiers, including Bugly. We also contacted all the developers of the vulnerable apps mentioned in this work. For Google and Bugly, we contacted them via their bug hunters' platform, as well as through email communication for remaining cases, except for `com.newsweekly.livepi`, where we utilized their in-app contact form due to the absence of an email address. Developer email addresses for Google Play apps were directly obtained from the Google Play Store. However, for Chinese apps, as contact information on the 360 Mobile Assistant website was unavailable, we relied on various sources, including the apps themselves, their respective websites, and development websites, to acquire email addresses. Throughout our communication with all involved parties, we meticulously outlined the identified vulnerability, its associated risks, and provided detailed information on potential remediation methods. Additionally, we offered app developers a fallback methodology to partially rectify their code if removing the insecure library proved challenging. Google acknowledged the issue after multiple interactions, stating they cannot address it "without introducing app compatibility risks". They also mentioned exploring the possibility of introducing warnings or errors to Android Studio when this behavior is observed, along with other mitigation methods. On the other hand, Bugly refused to accept our finding as an issue for their library. The remaining vendors did not respond to us (as of Feb. 21, 2024).

2 Related Work

In this section, we summarize related work from the literature, and discuss the unique aspects of our work.

TLS security analysis of mobile apps. In general, TLS problems identified in apps may involve two cases: defects in a given implementation such as the use of weak cipher suites [40], and improper use/choice/configuration of TLS libraries [51]. Fahl et al. [30] performed the first in-depth study of the widespread SSL problems through studying code snippets and advice in developer forums, and interviewing app developers. They revealed that customized SSL functions as used by developers, and their partial understanding of SSL are the two main reasons for weakening SSL security. Georgiev et al. [31] demonstrated that SSL certificate validation is completely broken in many security-critical apps and libraries, such as Amazon's EC2 Java library and Google AdMob, and attributed these vulnerabilities to badly designed APIs of SSL implementations, e.g., JSSE, OpenSSL and GnuTLS. Oltrogge et al. [35] discovered that Google Play failed to detect vulnerable TLS implementations in their investigation of the effectiveness of Google's Network Security Configuration (NSC) and Google Play Safeguards. Possemato et al. [36] performed an extensive study on Google network defense mechanisms and found that network security policy in apps can downgrade security by allowing cleartext protocols or by trusting the union of CAs from both System and User KeyStore. Additionally, they noticed that several popular ad libraries require apps to weaken their security policy.

Wang et al. [50,51] employed both static and dynamic analysis techniques to identify the vulnerable implementations of TLS in Android apps. Their tool, DCDroid [50], uncovered four types of vulnerable implementations of TLS (i.e., `X509TrustManager`, `HostNameVerifier`, `WebViewClient`, `X509HostnameVerifier`). However, it could only detect simple vulnerable TLS implementations (e.g., by implementing the standard Java interface with a few or even a single instruction), and may fail to identify complex ones. In practice, TLS functions can be improperly implemented in various ways, which does not affect our methodology. DCDroid also found some apps to be vulnerable because they invoked vulnerable third-party SDKs; however, the entity that was validating the certificate was not identified. Applicable to all the aforementioned studies, no fine-grained attribution was conducted (e.g., identifying the responsible parties for validation failures, and validation hijacking between parties).

System proxy and dynamic analysis evasion. Most existing dynamic analysis studies do not consider proxy evasion by Android apps (e.g., [33,41,44,51]), which may have affected the reported results, i.e., missing connections from such apps. However, two studies [25,37] addressed this issue by using API hooking. Nevertheless, this approach is also not fully reliable as many apps are becoming aware of dynamic binary instrumentation (DBI). In contrast, the use of eBPF in Marvin at the kernel level is impossible for the apps to detect without root access. With regard to dynamic analysis evasion/detection, NCScope [56], a hardware-assisted tool to scrutinize native code of potentially packed apps, leveraged ETM (a hardware feature of ARM processors) and eBPF to collect real execution traces and relevant memory data of apps. However, NCScope can only detect the presence of self-protection and anti-analysis mechanisms but not evade them (i.e., the app still crashes).

Third-party code identification. Pradeep et al. [38] conducted a comparative analysis of certificate pinning with an exclusive focus on third-party components in apps. To differentiate between first-party and third-party code, they set the following threshold: if a code path was found in more than five apps, it was categorized as a third-party source. Muslukhov et al. [32] attributed whether a package name corresponds to an app, a library, or a possible library, by searching an exhaustive list of package names for libraries and potential libraries; such lists of course become stale with time.

Sensitive data leakage. Wang et al. [49] uncovered a new attack vector: malicious libraries strategically target other vendors' SDKs integrated in the same host app to harvest private user data. They found 42 distinct libraries stealthily harvesting data from 16 popular SDKs, affecting over 19K apps. Nguyen et al. [33] performed a large-scale measurement of 86,163 Android apps to understand the current state of GDPR's explicit consent violation. They found that 24,838 apps share personal data without the user's explicit consent. Pourali et al. [37] implemented ThirdEye to detect privacy issues via non-standard and covert channels. They identified that 2887/12,598 (22.92%) apps used *custom encryption* (where data is encrypted by an app before being sent over the TLS channel) for network transmission and storing private content in shared device storage. Reardon et al. [42] found that Jiguang's SDK (a Chinese SDK) evasively monitors user's activities and collects user information, e.g., GPS location, identifiers (e.g., device serial, IMEI), and the names of the installed apps. Although we identified several highly popular libraries involved in TLS validation hijacking, we are unsure about their motives, or whether they are even aware of it.

Novelty. Compared to previous work, we have achieved fine-grained detection and attribution of TLS certificate validation issues. Notably, no past work attributed to the truly responsible party behind a TLS validation failure—i.e., no distinction was between the connection originators and faulty validators, which as we found can be different. More critically, we bring attention to and measure the prevalence of what we call certificate validation hijacking—a completely new phenomenon, unidentified by any past work. Accordingly, our fine-grained attribution cases with their measurements demonstrate the chaos of TLS certificate validation practices between multiple parties in the Android development cycle. We also leverage kernel-level instrumentation to handle DBI-aware apps, to maximize analysis coverage, which can help future dynamic analysis frameworks.

3 High-level System Design

In this section, we provide an overview of our design behind Marvin. The primary goal of Marvin is to analyze TLS certificate validation issues in Android apps, and attribute such issues to responsible part of an app (e.g., the app developer's code or a specific library included in the app). Specifically, we aim to examine TLS certificate validation issues in four different cases based on hosts. Hereafter, we refer to TLS connections to unique destination addresses with one or more of the four validation issues (see below) as *insecure connections*. We investigate the root cause of a validation failure or hijack, and attribute it to either the app or one of its libraries.

Selected TLS certificate validation issues. We consider the following four cases adapted from `badssl.com` to cover the most obvious checks for our analysis. Additional tests can be easily added to Marvin albeit at the expense of increased time requirement for analyzing each app. (1) *Unverified Certificate Signature*. The app accepts any received certificate without performing any signature validation, although it may still check the expiry date and domain name. (2) *Self-signed Certificate*. The app performs signature validation on the received certificate using the public key contained in the certificate itself. Nonetheless, it may still verify the expiry date and domain name. (3) *Expired Certificate*. The app performs secure domain and signature validation on the received certificate, but it disregards the validity time of the certificate. (4) *Domain Mismatch*. The app securely checks the expiry date and performs signature validation on the received certificate, but it skips matching the certificate domain and server names.

Threat model. We assume a network-based attacker, positioned anywhere between a faulty app (i.e., an app with a TLS validation error) and the server(s) it connects to, in line with the Dolev-Yao [27] threat model, i.e., the attacker can read/manipulate the network traffic. The attacker also needs to identify faulty apps before launching the attack, e.g., via simple TLS validation tests (being able to use mitmproxy could be enough). This attack can result in a wide range of consequences, including but not limited to session hijacking, credential/PII theft, phishing attacks, and malware/code injection. We assume no other capabilities for the attacker.

3.1 Attribution of Validation Issues

When certificate validation is not performed properly for a specific TLS connection, we would like to pinpoint where the issue likely stemmed from, unlike in previous studies (e.g., [46, 50]) where the issue was merely identified to have occurred in a monolithic app.

The common practice in Android app development involves incorporating third-party libraries (DEX/native) in developers' own code, producing a single app package, containing code from multiple parties. For instance, the "Fitdays" (`cn.fitdays.fitdays`) app with 1M+ downloads contains `com.tencent.bugly.proguard` (third-party code) package in its code, which improperly handles TLS certificate validation. In contrast, the "Playit" (`com.playit.videoplayer`) app, with 100M+ downloads, improperly modifies the default TLS verifiers, affecting TLS certificate validation in various third-party code, including Facebook, Unity3d, and AppLovin SDKs. It is important to note that the presence of a certificate validation issue may or may not be under the control of the app/library developer, depending on the specific code segment responsible for the issue. This distinction is crucial in determining the appropriate remedial actions.

Determination of third-parties. To the best of our knowledge, there exists no universal, accurate and automated way to partition an app based on vendors (e.g., not scalable if based on an exhaustive list [32, 45]). Therefore, we follow Pradeep et al. [38] (in their case for certificate paths), to consider any package name appearing in more than 5 apps as a third-party library, as opposed to code written by the app developer. Although a library can consist of multiple packages, which may or may not have common package names, using packages as an approximation for libraries is still valid since multiple packages can be seen as a way of modularizing a library. Additionally, if vendor information becomes available, package-level attribution can be aggregated to the vendor level. Likewise, we apply the same threshold for determining third-party domain names, i.e., if a domain is contacted by over 5 apps, we consider it to be a third-party domain name, as opposed to the app's domain.

Validation overrides and hijacking. Some Android app developers may override the standard Java interface (e.g., from classes `HostnameVerifier` and `X509TrustManager`) with custom code, e.g., to make use of certain TLS features (cf. [50, 51]). However, this customization may lead to one or multiple of the aforementioned improper validation cases. From our observation, TLS certificate validation is composed of: a hostname verifier to check for Domain Mismatch, and the rest of the validation often in the form of callbacks in the trust manager (including checks for the other three cases above). Correspondingly, this is reflected in the overrides, `HostnameVerifier.verify()` and `X509TrustManager.checkServerTrusted()`.

Aside from the aforementioned regular overrides, we observe that the validation override can be done in a way that *globally* affects all subsequent TLS certificate validations in the HTTPS protocol if they use the `HttpsURLConnection` class with default values for socket factory and hostname verifier; alarmingly, this phenomenon is prevalent (as observed in our results). Technically, Android provides two static public methods, namely `setDefaultSSLSocketFactory()` and `setDefaultHostnameVerifier()` to override the default TLS validation environment globally, stored statically in `HttpsURLConnection`, without any concurrency control. While these functions are well-documented and their use is not new, third-party libraries performing such global overrides can have severe security implications (e.g., bypassing certificate validation unbeknownst to the app developer). We note that this might be either intentional or just inadvertent.

```
private static HttpsURLConnection a(URL url, int i) throws NoSuchAlgorithmException,
KeyManagementExceiption, IOException {
    SSLContext sSLContext = SSLContext.getInstance("TLS");
    sSLContext.init(null, new TrustedManager[]{new b()}, new SecureRandom());
    HttpsURLConnection.setDefaultSocketFactory(sSLContext.getSocketFactory());}
    HttpsURLConnection.setDefaultHostnameVerifier(new a());
    HttpsURLConnection httpsURLConnection = (HttpsURLConnection) url.openConnection();
    return httpsURLConnection;
}
```

Listing 1: Global TLS certificate validation override in Tencent Bugly (a widely used library)

We define TLS certificate *validation hijacking* as when either an app or a library attempts to validate its certificate by invoking standard libraries or its own implementation, but the validation code is overridden by another party, which does not perform any validation checks, or lacks any required checks as defined in RFC 5280 [43]. Consequently, the certificate validation is ultimately carried out by the other party rather than the app or the library's code. Listing 1 shows an example where Tencent Bugly [17] (a popular Chinese library for exception reporting) performs global overrides in its ProGuard helper SDK (which is different from the Android's ProGuard optimizer and obfuscator); this causes certificate validation of the entire app, including other libraries, to be through Bugly's code, which turned out to be insecure as we found.

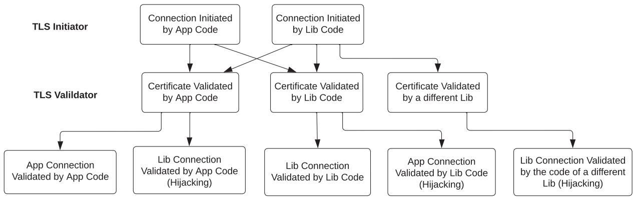

TLS Initiator
TLS Valildator
Connection Initiated by App Code
Connection Initiated by Lib Code
Certificate Validated by App Code
Certificate Validated by Lib Code
Certificate Validated by a different Lib
App Connection Validated by App Code
Lib Connection Validated by App Code (Hijacking)
Lib Connection Validated by Lib Code
App Connection Validated by Lib Code (Hijacking)
Lib Connection Validated by the code of a different Lib (Hijacking)

Figure 1: Overview of our attribution cases (except the racing hijacking case involving multiple parties)

Attribution cases. Based on the aforementioned observations, we consider our fine-grained attribution of the validation issues in six cases as follows; see also Fig. 1. Note that we consider (and have confirmed from the results) that the standard validation code is properly implemented and hence not reflected in the attribution (as no issues occurred).

1) App connection validated by app code. This refers to when the app developer uses custom code or overrides standard code to validate the certificate for connections initiated by the app developer, but does so improperly. Although insecure, the impact of this practice is limited to an app developer's connections alone, in a given app.

2) Library connection validated by library code. Similarly, a library included in the app can also perform certificate validation of connections initiated by itself, through either custom code or overrides. The impact of this case is also limited to an individual library, but considering a library is potentially used by many apps, the connections from that library will be vulnerable in all apps using the library.

3) App connection validated by library code (hijacking). Due to validation hijacking, once a library calls one or both of the set default functions, all Java-based HTTPS connections that utilize the default verifier within a given app will be validated by the override functions of this library, which the app developer may be unaware of. In this case, the victim connections are initiated by the app developer.

4) Library connection validated by app code (hijacking). Likewise, when the code calling the set default functions is written by the app developer, the victim connections can also be initiated by a library included in the app, which may not be anticipated by the library developer.

5) Library connection validated by the code of a different library (hijacking). In this case, both the victim connection and the code causing the global override are from libraries included by the app, but different libraries (e.g., Tencent improperly validating a connection's certificate initiated by Google).

6) Race condition with multiple callers (hijacking). The global override functions may also be called by multiple parties in the same app. As each call to set the default functions will affect certificate validation for all Java-based HTTPS connections that utilize the default verifiers, it will always be the last call that takes effect. As the consequence is determined by timing, aside from showing the presence of this issue, we do not measure the timings to find out who wins (which can vary across different runs).

3.2 Technical Challenges, Solutions, Overview

We highlight several unique technical challenges faced in the attempt to achieve our analysis objectives and our solutions.

Traffic separation/redirection. As our analysis target is a specific app, we must separate its traffic from that of other apps on the operating system. By default, each network connection does not necessarily have any identifier specific to an app. Moreover, we encounter a challenge where certain apps (almost 70% of apps from the Chinese store) disregard the proxy configuration set in the device settings for some of their connections. Instead, they establish direct connections to their intended destinations, bypassing the designated proxy server. To address these issues, we need a mechanism capable of identifying such connections, linking them to their originating apps, and redirecting them to our proxy server. In contrast to other approaches that either overlooked [33, 44] this issue or relied on VPNs [41, 51] or API hooking [25, 37], we leverage eBPF [9], which operates at the kernel level (requiring no changes to the kernel and remaining undetectable by apps), to intercept system events. Specifically, we utilize eBPF to hook into the Linux kernel's `cgroup` feature, enabling us to attach our instrumentation to specific resource allocation events within the OS. For network connections, we use the programs `connect4` (for IPv4) and `connect6` (for IPv6), corresponding to TCP socket creation in the Linux kernel.

To achieve traffic separation, considering that each Android app is assigned a unique user ID, we can trace the origin of each socket by logging the user IDs associated with their creation through the `cgroup` feature. We opt for user IDs over process IDs due to the possibility of an app having multiple dynamically generated process IDs.

For traffic redirection, we employ `eBPF` to create a hook in the `cgroup` feature, intercepting, logging, and modifying socket destinations during resource allocation, especially for HTTP/S ports. This interception occurs if the destination address does not match that of our proxy server. By leveraging the user ID of the app, this approach allows us to selectively forward app traffic in a granular manner. Although this methodology requires root access, alternative methods are not sufficient for our task. For instance, `iptables` which also requires root access, does not support the log feature in Android, which is used to enumerate the connections. VPNs do not require root access but can be easily detected [20], and traffic capturing from the WiFi access points does not allow us to perform traffic separation.

Dynamic analysis evasion. We also observe that a significant number of apps employ robust evasion techniques to counter dynamic analysis, particularly function hooking. This means the app being analyzed will crash (among other misbehavior) once hooking is detected, thus prematurely terminating our analysis. This behavior is known as dynamic binary instrumentation (DBI) detection. There have been a number of studies on "Android packers" [23, 28, 56], with academically proposed "unpackers" [53–55], most of which tend to be no longer up-to-date (packing and unpacking are like an arms race). Note that packers can usually detect rooting, emulation, debugging, and function hooking with a highly obfuscated native library (`.so`). We find a significant portion of the packed apps making use of very strong commercial Android packers, for which we are not aware of any unpackers (academic or commercial), to the best of our knowledge; example packers include Ijiami [11], Bangcle [6], Netease [16], 360 [2], and Ali [3], among many others. For instance, the Ijiami packer employs direct syscalls (the assembly instruction) to examine the app's memory maps for any signs of tampering by offset calculation. Hence, app-specific tweaks are necessary to avoid detection (which is manual and not scalable). To maximize our analysis coverage, we discuss how we deal with DBI detection (included in most packers) in Section 4.2.

Overview of Marvin. In brief, we run the target app by interacting with its UI in an automated manner to trigger as many connections as possible, and meanwhile intercept its traffic with mitmproxy [15] (Sec. 4.2 under "TLS Interception") with invalid certificates. We prepare certificates corresponding to the aforementioned 4 validation issues (Sec. 3) for the proxy, and thus any connections ending up in the intercepted traffic (meaning problematic certificates accepted) will match with the improper validation in the app's code. In parallel, we employ dynamic analysis via eBPF for traffic separation/redirection (Sec. 3.2 under "Traffic separation/redirection") and Frida hooking [34] to record the stack traces of a pre-determined set of functions in the apps to serve two purposes: i) to aid in the attribution analysis of the certificate validation issues; and ii) to gain a better understanding of what is sent in the HTTPS (including custom-encrypted) traffic. We further explain the implementation details in Sec. 4.

4 Implementation of Marvin

Our analysis with Marvin is conducted in two phases: 1) execution and data collection; 2) analysis and attribution; see Figure 2 for an overview. We utilize Python and JavaScript to implement the detection of TLS certificate validation vulnerabilities and fine-grained attribution (approx. 3.3K LOC). We used Python script to create TLS passthrough and certificate exchange features, which were seamlessly integrated into mitmproxy as addons. This allowed us to intercept and store only insecure TLS connections. For traffic separation and redirection, we leveraged eBPF programs as explained in Sec. 3.2, developed in C. For the dynamic analysis, i.e., hooking network and certificate validation functions, we used Frida for which hooks were written in JavaScript. Moreover, we made modifications to ThirdEye [37] to manage the orchestration of app executions and detect custom encryption

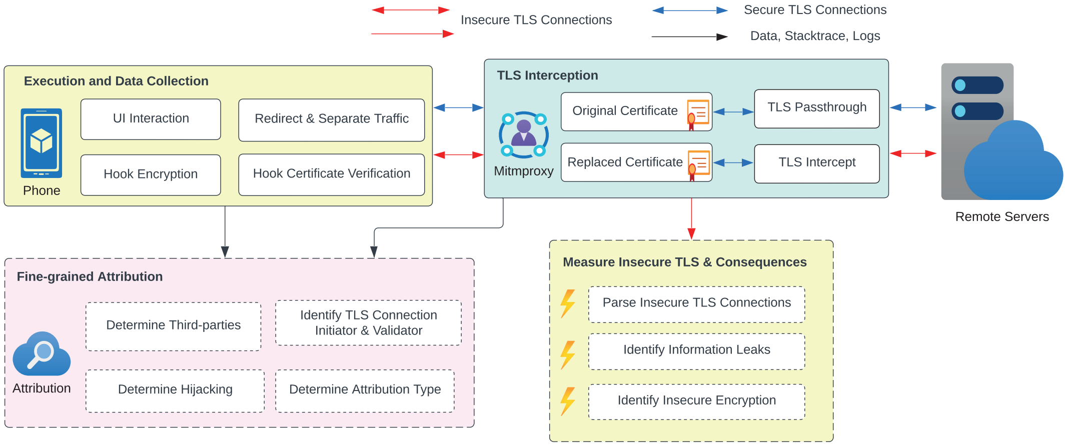


Figure 2: Components and workflow of Marvin

channels. Finally, we developed Python scripts to perform our analysis over the collected data.

4.1 Preparing the Analysis Environment

We first explain several considerations for preparing the environment and one-time actions before detailing the execution, data collection, and data analysis phases.

Root detection evasion. As mentioned in Sec. 3.2, the state-of-the-art commercial packers also provide features to detect and evade dynamic analysis. However, among these features, we must address root detection, as our method requires root permissions to set up the dynamic analysis tools, such as eBPF and Frida. To obtain root access, we utilized Magisk [14], and configured it to run as part of the Zygote process (a system-level process in Android that serves as a parent process for all app processes). Marvin scans the `AndroidManifest.xml` file of each installed app to automatically extract their package names and services, which are then added to the Magisk denylist. This denylist prevents apps and services from accessing the Zygote process and any associated root privileges, which reduces the chance of root detection.

Attaching and adapting eBPF programs. We utilized the *bpftool* tool within the *eadb* [8] environment (a Debian-based shell environment to run BCC, bpftrace, and bpftool on Android) to attach our *eBPF* programs to the kernel. However, *eadb* lacked native support for *cgroup* that we used, prompting us to enhance it by adding functionality for cgroup-based eBPF programs. Subsequently, we packaged our programs into a Magisk service module, enabling automatic execution after device boot, which allowed Marvin to maintain the continuous operation of our eBPF programs on the devices, facilitating the separation and redirection of app network traffic.

4.2 Execution and Data Collection

In this phase, Marvin automatically installs and executes the selected apps on configured Android devices, and then runs a mitmproxy for each of four types of invalid certificates. At this point, the eBPF programs are already running (as discussed above), and certain functions of the target apps (mainly for attribution) are hooked using Frida (with Javascript code). Finally, Marvin starts the user interaction simulator on the phone to explore the apps and trigger network connections. We discuss individual aspects of this phase below.

UI interaction. To maximize execution path coverage to trigger network connections, Marvin needs to simulate a user's interaction with the app's UI, e.g., tapping and entering text. To this end, we utilized the ThirdEye UI interaction module and improved its interaction speed by adjusting the timing parameters, and updating its duplicate element detection approach, wherein each UI item is interacted with twice to induce the app to create connections again in case the certificate validation was correct (hence connection aborted), as explained below under "Inducing apps to re-establish connection".

TLS interception. The key to our analysis of certificate validation issues is the ability to intercept the TLS traffic, replace the certificates, and observe how the app reacts. Therefore, Marvin launches mitmproxy and replaces the server's certificate with four types of misconfigured certificates corresponding to the four types of validation issues, for each server contacted by the app. If the proxy server receives a TLS error about validation failures (which is expected from the app), Marvin adds the server's SNI and IP address to a whitelist, and the proxy works as an SNI server without replacing the certificate. If the certificate is accepted without errors, Marvin flags it as improper validation (hence an insecure connection).

Certificate generation and replacement. Marvin generates the four types of misconfigured certificates as discussed in Sec. 3 per connection on-the-fly using the OpenSSL library, except for the case of testing domain mismatch. This approach avoids the mixing of results from across different validation issue types. For instance, if a self-signed certificate contains incorrect Subject Alternative Names (SANs), it is difficult to distinguish if the rejection is due to domain mismatch or the use of a self-signed certificate. Marvin only imports our trusted root certificate on the phone for testing the case of expired certificates. Additionally, we generate a valid certificate using Let's Encrypt for a test domain to check domain mismatch scenarios. This approach helps us achieve greater coverage if an app uses a hardcoded root certificate store instead of that of the OS (cf. `rustls` library).

Reverting to original certificate for correct validation. In the case of correct certificate validation, our replaced certificate will be rejected leading to a failed connection, which may affect the app's functionality, preventing us to see subsequent potentially improper validation. To address this issue, we rely on TLS protocol alert messages to identify failures due to correct TLS certificate validation. We developed an add-on for mitmproxy that utilized the `tls_failed_client` event hook, capturing TLS connection data when the proxy server received a TLS error from the client. In this case, we whitelist the combination of server IP address and TLS Server Name Indication (SNI) and forward the connection to the SNI proxy without modifying the original certificate.

Proxy implementation. Marvin utilizes three types of proxies for TLS interception, namely HTTP forward, HTTP transparent, and SNI proxy, leveraging mitmproxy for their implementation. When apps respect the system proxy, they automatically utilize the forward proxy, which can capture both their original HTTP method and the CONNECT method. However, certain apps do not adhere to the system proxy settings and establish direct connections with their server. As mentioned in Sec. 3.2, these connections are forcefully forwarded to the server, but the apps do not use the CONNECT method. This makes reconstructing the original HTTP messages to the CONNECT method challenging. To overcome this issue, we enable the transparent proxy functionality mode in the proxy server, which resolves the problem by modifying the destination socket address without reconstructing the HTTP messages. We also incorporate the SNI proxy functionality, which enables Marvin to redirect connections that undergo TLS certificate validation appropriately without interception.

Finally, our data collection includes all insecure HTTPS connections obtained either through the proxy server or redirected from eBPF, encryption/decryption data from ThirdEye, SDK certificate validation functions and device information (e.g., MAC address, phone number, and device email).

Inducing apps to re-establish connection. As mentioned earlier, if the app rejects the certificate, Marvin reverts to the original certificate for subsequent connections. However, the app may not automatically attempt to re-establish the connection after the failure. To address this issue, we modify our UI interactor to interact with each UI object twice, potentially triggering a re-establishment of connections caused by a UI event. If the interactor does not discover any new UI elements to explore, which could be the cause of the failed reconnection, the interactor relaunches the app to check again.

Handling DBI-aware apps. Although we are unable to defeat most commercial packers as mentioned in Sec. 3.2, to maximize analysis coverage, we have implemented a fallback mode to handle DBI detection. This mode involves performing a preliminary check to determine if the app crashes/freezes upon opening. If a crash/freeze is detected, we restart the analysis with our DBI tool (i.e., Frida) disabled. We utilize the `logcat` command and the corresponding user ID of the app to identify such crashes. Additionally, we monitor the appearance of non-responsive windows by checking the `mCurrentFocus` value through the `dumpsys activity activities` command. The presence of a non-responsive window indicates that the app has become frozen. This way, Marvin can automatically continue the analysis of these apps at the cost of losing fine-grained attribution.

4.3 Analysis and Attribution

In this section, we discuss the approaches Marvin applied to collect and process data from the execution phase and to perform attribution of certificate validation issues.

Identification of certificate validation issues. To filter out traffic from a target app, Marvin compares the network records obtained through eBPF with the insecure TLS connection records from the proxy for each misconfigured certificate type during the data collection phase. If any insecure connection from the app is detected, we perform the attribution process to determine the origin of the improper certificate validation.

Identification of certificate validation origins. At this point, Marvin has obtained per-app insecure connection information with certificate validation issues, but this is only from the network perspective (i.e., traffic). The next step for Marvin is to hook the specific functions responsible for the certificate validation issues for fine-grained attribution. Rahaman et al. [39] introduced a static analysis tool for detecting cryptographic vulnerabilities in Java, which demonstrated various cryptographic API misuses. Inspired by their approach, we have identified that the `checkServerTrusted()` (in the `X509TrustManager` class) and `verify()` (in the `HostnameVerifier` class) interfaces, and the `createSocket()` method (in the `SSLSocketFactory` class) can be misused for improper certificate validation. Hence, to obtain stack traces and certificate information, we used Frida to create hooks for these functions and integrated them with Marvin. It is worth noting that Java interfaces cannot be individually hooked; instead, their implementations must be employed for hooking. To identify the implementations of the certificate verifier interfaces, we first leveraged Frida's enumeration feature to list all methods with the same function signature within the app. Subsequently, we employed Java reflection to confirm that their declaring class matches that of the interface's class.

Mapping a TCP connection to a TLS session is challenging as our target functions work at different semantic levels, e.g., the `verify()` function does not have access to any socket information. To address this issue, we start our flow from `createSocket()` and record the socket information. Then, in `checkServerTrusted()`, with access to both certificate information and socket information, we create a hash of the certificate chain serials and map it with the socket information (the certificate serials are different for each connection due to on-the-fly certificate generation; see Sec. 4.2 under "Certificate generation and replacement"). Finally, in `verify()`, we can extract the SNI name, and we re-calculate the hash of the certificate chain serials and check if they match with the socket information. This allows our tool to accurately trace and map the connections and their corresponding TLS certificate validation functions.

Performing attribution. We examine the stack traces to check if the entity creating the TLS connection (highlighted in blue in the stack traces in Listing 2 to Listing 7) is the same entity executing the validation function (highlighted in red in the listings, if different). The lines in between are from standard network libraries (e.g., OkHttp) and thus the pattern is straightforward to parse. We provide a list of stack traces for each type of attribution from Listing 2 to Listing 7. In Listing 2, the entity `com.datayes.common` triggered the TLS certificate validation (blue, bold, line=5) and also performed the actual validation (blue, bold, line=1), hence, this case being classified as "app connection validated by app code" (not hijacked);

Listing 3 highlights the same issue for library code. Listing 4 shows a validation hijacking case where `com.dnurse.main.ui` created the TLS connection, while `com.tencent.bugly.proguard` invoked `checkServerTrusted` to perform the certificate validation. Therefore, this case is categorized as "app connection validated by library code" (hijacking). A reversed hijacking case is shown in Listing 5. Listing 6 demonstrates a case where both the victim and improper actor are libraries (but different ones). The package `com.umeng.commonsdk`, a popular library, established a TLS connection with its remote server, but the certificate validation was performed by `com.kuaishou.weapon` (another popular library). Listing 7 shows a scenario where multiple certificate validation hijackings occurred (but the actual consequence may depend on execution timing); here, the certificate validation was initially done by Tencent Bugly, followed by Baidu which overrode the default verifier thereafter.

```
1  at com.datayes.common.net.interceptor.ssl.OkHttpSSLSocketFactory$1.checkServerTrusted(Native Method)
2  at com.android.org.conscrypt.Platform.checkServerTrusted(Platform.java:260)
3  at com.android.org.conscrypt.ConscryptEngine.verifyCertificateChain(ConscryptEngine.java:1638)
4  ...
5  at com.datayes.common_cloud.net.interceptor.TokenInterceptor.intercept(TokenInterceptor.java:97)
```

Listing 2: App connection validated by app code

```
at cn.jiguang.net.DefaultHostVerifier.verify(Native Method)
at com.android.okhttp.internal.io.RealConnection.connectTls(RealConnection.java:200)
at com.android.okhttp.internal.io.RealConnection.connectSocket(RealConnection.java:153)
...
at cn.jiguang.net.HttpUtils.a(Unknown Source:196)
at cn.jiguang.net.HttpUtils.httpPost(Unknown Source:1)
```

Listing 3: Library connection validated by library code

```
at com.tencent.bugly.proguard.s.checkServerTrusted(Native Method)
at com.android.org.conscrypt.Platform.checkServerTrusted(Platform.java:260)
at com.android.org.conscrypt.ConscryptEngine.verifyCertificateChain(ConscryptEngine.java:1638)
...
at com.dnurse.main.ui.FlashActivity.downLoadImage(FlashActivity.java:11)(SourceFile:341)
at com.dnurse.main.ui.FlashActivity$a.doInBackground(FlashActivity.java:1)
```

Listing 4: App connection validated by library (hijacking)

```
at rich.y$a.verify(Native Method)
at com.android.okhttp.internal.io.RealConnection.connectTls(RealConnection.java:200)
at com.android.okhttp.internal.io.RealConnection.connectSocket(RealConnection.java:153)
...
at com.growingio.android.sdk.data.net.HttpService.performRequest(HttpService.java:132)
at com.growingio.android.sdk.data.net.HttpService.performRequest(HttpService.java:81)
```

Listing 5: Library connection validated by app (hijacking)

```
at com.kuaishou.weapon.p0.q2$a.checkServerTrusted(Native Method)
at com.android.org.conscrypt.Platform.checkServerTrusted(Platform.java:260)
at com.android.org.conscrypt.ConscryptEngine.verifyCertificateChain(ConscryptEngine.java:1638)
...
at com.umeng.commonsdk.statistics.internal.c.a(Unknown Source:170)
at com.umeng.commonsdk.statistics.internal.c.a(Unknown Source:57)
```

Listing 6: Library connection validated by another library (hijacking)

```
/* ———————— (1) Baidu is hijacked by Bugly ———————— */
at com.tencent.bugly.proguard.s$1.checkServerTrusted(Native Method)
at com.android.org.conscrypt.Platform.checkServerTrusted(Platform.java:260)
at com.android.org.conscrypt.ConscryptEngine.verifyCertificateChain(ConscryptEngine.java:1638)
...
at com.baidu.lbsapi.auth.g.a(Unknown Source:47)
at com.baidu.lbsapi.auth.g.a(Unknown Source:30)
/* ——————— (2) Baidu is validated by Baidu again ——————— */
at com.baidu.location.h.p.checkServerTrusted(Native Method)
at com.android.org.conscrypt.Platform.checkServerTrusted(Platform.java:260)
at com.android.org.conscrypt.ConscryptEngine.verifyCertificateChain(ConscryptEngine.java:1638)
...
at com.baidu.location.h.l.run(Unknown Source:171)
```

Listing 7: Multiple hijacking actors (race condition)

Detecting information leaks. We collect traffic from the affected communications due to validation issues, including hijacking. We then proceed to perform deep packet inspection, a detailed examination of network traffic that includes the analysis of both HTTPS headers and bodies, to identify information leaks that may have occurred due to insecure TLS certificate validation, as well as extracting potential custom encryption channels (additional encryption on top of a standard protocol) using the data from ThirdEye [37].

5 Results

Experimental setup and app dataset. We run the selected apps on two Pixel 7, and one Pixel 6 phone with Android 13 factory images, and the analysis scripts on three desktop PCs with Ubuntu 22.04 (Intel Core i7-10700 CPU, 48GB RAM, and two i7-8700 CPUs with 16GB RAM).

For the dataset to represent popular Android apps, we download top-rated apps from 360 Mobile Assistant (Qihoo 360 Appstore, a popular Chinese app store [1]) for Chinese apps, and Google Play apps from Google Play store based on the rank of APKPure [4], covering a variety of categories, such as online shopping, entertainment, travel, office tools, and finance. These apps are distinct and demonstrate unique characteristics from the perspective of both functionality and security [48]. We eventually collected a total of 4121 Chinese apps and 5452 Google Play app as our dataset to start with.

Marvin executed each app four times, with each run dedicated to testing a specific certificate problem. For each run, it typically required around three minutes to explore and interact with UI elements (i.e., around 12 minutes per app). However, if custom encryption is detected during a certificate validation test, Marvin will run the app twice for this specific certificate test in order to detect insecure custom encryption. In such scenarios, Marvin executes the apps between 4–8 times, varying based on the detection of custom encryption. Each app took approximately 12–24 minutes to conduct a comprehensive analysis of the four types of certificate validation issues and identify any potential insecure custom encryption.

During the testing phase, we observed that 2096/4121 (50.9%) of Chinese apps experienced crashes (either handled by their developers or not), and 948 of them did not generate network traffic, while 2147/5452 (39.4%) of Google Play apps crashed, 275 of them without network traffic. After we disabled Frida, 329/948 (34.7%) Chinese apps and 163/275 (59.3%) Google Play apps started to connect to their remote servers, indicating the presence of anti-hooking mechanisms in these apps. We were unable to attribute all the remaining crashes to a specific cause on a per-app basis. Examples of speculated causes include root detection, certificate pinning, and device incompatibility. We also observed that 408 of the Chinese apps and 140 of the Google Play apps generated no network traffic despite no crashing. Hereafter, we exclude the apps without observed network traffic from reported results, and consider the adjusted totals of 2765 and 5061, for Chinese apps and Google Play apps, respectively.

5.1 Certificate Validation Issues

We found that 1529/2765 (55.3%) of Chinese apps were identified to have at least one of the four certificate validation issues; in contrast, the percentage of insecure Google Play apps was much lower: 322/5061 (6.4%). In terms of each certificate validation issue, for Chinese apps, 1307/2765 (47.3%) were identified with unverified certificate signatures, 1375/2765 (49.7%) trusted self-signed certificates, 1310/2765 (47.4%) accepted expired certificates, and 1059/2765 (38.3%) ignored mismatched domain in the certificate validation. For Google Play apps, 244/5061 (4.8%) were identified with unverified certificate signatures, 243/5061 (4.8%) trusted self-signed certificates, 231/5061 (4.6%) accepted expired certificates, and 236/5061 (4.7%) ignored mismatched domain in the certificate validation. We also grouped the apps based on the number of insecure TLS connections, e.g., apps with 1 to 10 insecure TLS connections, 11 to 20, 21 to 30, and over 30; see Table 1. In the four certificate test cases, Chinese apps established a significantly higher percentage of insecure TLS connections; see Table 5 (in the appendix).

We observe that 1) the percentage of apps from the Chinese app store possessing certificate validation issues is notably higher compared to Google Play apps; 2) the proportion of apps in both Google Play and the Chinese app store with each validation issue does not show a significant difference; and 3) for both the affected number of apps and the resulting insecure connections, Chinese apps are significantly higher than Google Play apps, primarily due to the use of a small number of highly popular but faulty libraries in Chinese apps.

5.2 Attribution Results

Attributed insecure apps. We were able to attribute the insecure connections of the four certificate validation types to 1014/1529 (66.3%) Chinese apps and 139/322 (43.2%) Google Play apps; see Table 2. We also observe that 1) for the non-hijacking cases, improper validation happens more to libraries in Chinese apps (48.9%), while for Google Play apps it is more caused by the app code itself (30.7%). This leads to more overall apps being affected in the Chinese app store (as a popular faulty library can make many apps vulnerable).

Attributed insecure TLS connections. We first group the parties (app vs. library code) initiating the insecure TLS connections. For Chinese apps, the average percentage of insecure TLS connections created by libraries is 76.5% across four certificate test cases, while the connections initiated by apps themselves (i.e., app code) account for 23.5%; this trend is reversed in Google Play apps (16.2% from libraries and 83.8% from apps, see Table 8 in the appendix). We then identify the entities responsible for these insecure TLS connections, i.e., which code performs the insecure validation. We were able to attribute 68.4% insecure TLS connections in Chinese apps (10.0% were from app code, and 58.4% were from libraries), see Table 3. In contrast, we could attribute 44.5% of insecure TLS connections for Google Play apps (21.0% from app code, and 23.6% from libraries). We could attribute more Chinese apps due to the prevalence of a few faulty libraries (which we could attribute) in them. Moreover, in Google Play apps, the percentage of insecure TLS connections caused by faulty library validators was merely 7.7%, while faulty validators in app code led to 36.8% of insecure TLS connections.

Libraries involved in hijacked validation. We identified the libraries that call `setDefaultSSLSocketFactory()` and `setDefaultHostnameVerifier()` with insecure validation functions, and thereby, hijack certificate validation; see Table 4. For example, Bugly (`com.tencent.bugly`) overrides the default SSL socket factory through `setDefaultSSLSocketFactory()`. This results in hijacking Android OkHttp's TLS connections for non-Bugly TLS connections in apps that utilize Bugly. ByteDance SDK (`com.bytedance.sdk`) [7] is another popular library, designed to log events in mobile apps and send them to TikTok for targeted ads, measurement,

Certificate Type	**Chinese Apps (#apps, % of apps)**				**Google Play Apps (# apps, % of apps)**			
	[1, 10]	[11, 20]	[21, 30]	>30	[1, 10]	[11, 20]	[21, 30]	>30
Unverified Certificate Signature	910 (32.9%)	214 (7.7%)	95 (3.4%)	88 (3.2%)	208 (4.1%)	20 (0.4%)	4 (0.1%)	12 (0.2%)
Self-signed Certificate	971 (35.1%)	216 (7.8%)	99 (3.6%)	89 (3.2%)	208 (4.1%)	21 (0.4%)	5 (0.1%)	9 (0.2%)
Expired Certificate	938 (33.9%)	206 (7.5%)	83 (3.0%)	83 (3.0%)	200 (4.0%)	14 (0.3%)	9 (0.2%)	8 (0.2%)
Domain Mismatch	805 (29.1%)	151 (5.5%)	64 (2.3%)	39 (1.4%)	216 (4.3%)	9 (0.2%)	6 (0.1%)	5 (0.1%)
Total	1287 (46.5%)	406 (14.7%)	195 (7.1%)	128 (4.6%)	298 (5.9%)	32 (0.6%)	14 (0.3%)	15 (0.3%)

Table 1: The number and percentage of apps that established insecure TLS connections (grouped in different ranges) in each type of certificate validation issue among Google Play and Chinese apps. Total denotes the number and percentage of unique apps that possess one or multiple of the four types (de-duplicated).

Attribution Type	**#App (CHN)**	**#App (GP)**
App connection validated by app code	361 (23.6%)	99 (30.7%)
Library connection validated by library code	747 (48.9%)	28 (8.7%)
App connection hijacked by library code	102 (6.7%)	12 (3.7%)
Library connection hijacked by app code	194 (12.7%)	50 (15.5%)
Cross-library hijacking	360 (23.5%)	23 (7.1%)

Table 2: The fine-grained attribution results; percentages are calculated over all insecure apps; App (CHN) denotes Chinese apps; App (GP) denotes Google Play apps. Note that the same app may be counted under more than one attribution type.

and conversion optimization; this SDK overrides the default SSL socket factory using `setDefaultSocketFactory()`. Developers also often incorporate the Baidu Location SDK (`com.baidu.location`) [5], to leverage Baidu services to access accurate location data and enable location-based features within their apps. Similar to Bugly, Baidu SDKs also override the default SSL socket factory through `setDefaultSSLSocketFactory()`. Note that the Bugly SDK is also found in 12 Google Play apps in our dataset.

Race conditions in certificate validation. We noticed that in 417 (27.3%) insecure Chinese apps, multiple parties (e.g., two libraries, or an app and a library) did the override, leading to uncertainty about validation logic, determined by timing. Specifically, the effective validation function relies on who set the default SSL Socket Factory or default Hostname verifier the last, right before a given validation call. For example, in `com.lingan.yunqi`, we observed that `com.kepler.sdk` called `setDefaultSSLSocketFactory()` first, and consequently validated the certificate of `ap1.qiyukf.com`. Later, Bugly (`com.tencent.bugly`) invoked `setDefaultSSLSocketFactory()`, and thus hijacked the validation for `baichuan-sdk.alicdn.com` and `baichuan-sdk.taobao.com`. We also observed apps in which both `setDefaultSSLSocketFactory()` and `setDefaultHostnameVerifier()` functions were invoked. For instance, in the case of `com.mobivans.onestrokecharge`, Bugly modified the default SSL socket factory by invoking `setDefaultSSLSocketFactory()`, which validated the certificate of the host `i.sdkyounger.com`. Subsequently, `com.kuaishou.weapon` called `setDefaultHostnameVerifier()`, and used its faulty verifier for `android.bugly.qq.com`, `i.sdkyounger.com`, `ulog.umeng.com`, etc.

5.3 Information Leaks

We perform deep packet inspection to detect information leaks resulting from insecure TLS connections vulnerable to MITM attacks, due to improper certificate validation. In the Chinese app dataset, we found that 1358/1529 (88.8%) of the apps transmit sensitive information using insecure TLS connections. Among them, 1354 (99.7%) use plain insecure TLS, 453 (33.4%) use custom encryption on top of insecure TLS, and 163 (12.0%) use weak/broken custom encryption on top of insecure TLS. For the Google Play dataset, we observed that 278/322 (86.3%) of the apps transmit sensitive information through plain insecure TLS connections. We group the leaked information into five distinct categories: Device, Network, Network Location, Location, and User Assets (see Table 7 in the appendix). We noticed that some apps use custom encryption to transmit user data. However, it is unclear whether the developers are aware of the TLS validation issues and thus attempted to (poorly) mitigate such issues, or if they are trying to conceal their activity on the network.

6 Case Studies

To further demonstrate the severity and practicality of the identified TLS certificate validation hijacking instances, we selectively analyze several apps (from different categories) for their susceptibility, and implement the exploits on our own devices. We group the exploit consequences to facilitate the discussion here. Note that for all the account takeover, impersonation, phishing attacks, and PII disclosure, we used our own devices for both attacker and victim devices on our local WiFi network, and our own test accounts and PII—i.e., the example attacks did not affect any user or provider. It is also reflected in all the studied cases that the victim party (e.g., end-users, app developers, identity providers, ad networks) is *unaware of such hijacking and not to blame*. We summarize our attacks and consequences in Table 6 (in the appendix).

Account takeover and impersonation. Android apps support various methods for user authentication, including app-specific login, and the use of third-party identity providers (IdPs) like Facebook, X/Twitter, and Google. Compared to affecting a single app, we have found a more worrisome fact that the SDKs used to interact with such IdPs can also be affected by certificate validation hijacking. For example, Di-

Connection	Validation Code	Hijacked	Attribution	# Connection (Chinese)	# Connections (Google Play)
App connection	App code	No	AppConnectionValidatedByAppCode	3840 (7.8%)	1311 (19.9%)
Lib connection	Lib code	No	LibConnectionValidatedByLibCode	12,531 (25.4%)	266 (4.0%)
App connection	Lib code	**Yes**	AppConnectionHijackedByLibCode	1075 (2.2%)	72 (1.1%)
Lib connection	App code	**Yes**	LibConnectionHijackedByAppCode	4623 (9.4%)	1118 (17.0%)
Lib connection	Another Lib's code	**Yes**	LibConnectionHijackedByLibCode	11,598 (23.6%)	170 (2.6%)

Table 3: The fine-grained attribution result of insecure TLS connections

voom (`com.divoom.Divoom`, available on Google Play for pixel art editing, 500K+ downloads) employs X/Twitter and Facebook for user authentication, in addition to its independent login component, and uses Bugly for bug reporting. Notably, Bugly replaces the default TLS certificate verifier with its insecure verifier upon app launch. Divoom's independent login component utilizes the Square OkHttp implementation, ensuring its login form remains unaffected. However, it relies on FBAndroidSDK [10] and Twitter4J [19] libraries to interact with the IdPs, which are dependent on Android's OkHttp implementation, and thus vulnerable to Bugly's validation hijacking. We managed to eavesdrop on the OAuth traffic (i.e., MITM), and get the OAuth access token, and and extract user data from the corresponding X/Twitter (e.g., Tweets, lists, collections, profile information, and account settings) and Facebook (e.g., user email) accounts, as well as access the user's Divoom account.

Apart from SSO services, we also observed some apps' independent logins being affected by such hijacking and thus exposed credentials such as (hashed) long-term/one-time passwords. For instance, among the apps from the Qihoo appstore, `asia.share.superayiconsumer` is affected by the Baidu location service; and `com.guixue.m`, and `ch999.app.UI` are affected by Bugly.

Several apps also undermine their own login pages by customizing TLS validation functions. Such apps include: `cn.yonghui.hyd`, `android.jianzhilieren`, and `cn.mopon.film.xflh`. On the other hand, `com.belugaedu.amgigorae` uses a secure login page, but it fetches certain parts of its content remotely as HTML code without proper certificate validation, which we exploited for a demo phishing attack (see Fig. 3 in the appendix).

Ad modification and phishing. Ad networks/platforms typically use HTML5 to load and render ad content. Given the prevalence of ads on Android and the ease of modification of HTML, such content becomes an attractive target for attackers seeking to launch phishing attacks by modifying ads to mimic the app's legitimate login form. We found that the majority of ad platforms in the Google Play Store dataset, including AppLovin, Amazon Ads, and Facebook Ads, employ the Android OkHttp library, making them vulnerable to the TLS certificate verifier override.

For instance, *Paint by Number: Coloring Game* (`com.paint.bynumber.color.coloringgames`, 100M+ downloads), employs Bugly for bug reporting, and utilizes AppLovin, Facebook Ads, and Amazon Ads for displaying ads. By exploiting the validation hijacking by Bugly, we successfully manipulated the ad content loaded by AppLovin (from the network) for a phishing attack (see Fig. 4 in the appendix). Conversely, as discussed earlier, overriding the default TLS certificate verifier can also be performed by the app itself and can affect other third-party SDKs and libraries within it. For example, Playit (`com.playit.videoplayer`, a video player app with 100M+ downloads) uses various ad SDKs, including Flat Ads, Moloco, Smaato Ads, and AppLovin, among others. This app's code overrides the default TLS certificate verifier with an insecure one, affecting all the ads SDKs, opening the door to phishing attacks the same way (as in Fig. 4).

Remote code execution. Surprisingly, we observed that 53 apps in the Chinese dataset transmit Dalvik Executable files (dex files) over TLS connections hijacked by Bugly (and thus made vulnerable to MITM). 17 of these apps transmit Dalvik data to their servers, while 41 apps receive such data (five apps perform both). Such transmission of executable code over insecure connections exposes the apps (and perhaps the app servers) to code injection attacks, allowing malicious actors to manipulate the behavior of the app/server (cf. [52]). We randomly selected a few apps for manual analysis to understand the impact of this phenomenon. For `com.android.tutuerge`, a heavily packed app with 10M+ downloads, the transmitted binary is used to load a video player. For verification, we modified the transmitted binary using Apktool and the app performed no integrity checks before loading the binary into memory. This phenomenon can have significant implications for the security and privacy of user data within the app; e.g., to abuse the app's permissions to collect privacy-sensitive data, place a keylogger, and compromise user credentials.

Sensitive information leakage. Apps that access various sensitive information from the phone and send it to their remote endpoints, may expose sensitive PII to network attackers due to certificate validation hijacking. Examples include: `asia.share.superayiconsumer` exposes a user's GPS location due to TLS validation hijacking by Baidu (com.baidu.location); `cn.com.jschina.news` leaks GPS location, contact names, phone manufacturer, operator, and installed packages due to Bugly validation hijacking; and `com.newsweekly.livepi` exposes device fingerprinting information due to validation hijacking by Alibaba log service.

7 Prevalence, Root Cause, Mitigation, and Limitations

Prevalence of certificate validation function override. In the dynamic analysis, we observed 519/2765 (18.8%) Chinese apps, and 152/5061 (3.0%) Google Play apps involved default validation function override regardless of whether it is secure or not. Among them, 391/519 (75.3%) of the Chinese app cases and 17/152 (11.2%) of the Google Play cases were insecure, caused by a third-party library.

Dynamic analysis is subject to limited code coverage by nature, e.g., even though our UI instrumentation is quite comprehensive, we still may miss validation issues for TLS connections that are initiated only after login (as our login coverage is only partial). To better understand the prevalence of such insecure override functions in apps, we conducted a static analysis using Androguard [21] on our dataset. This analysis is aimed to cover override functions in both first-party and third-party code that were not triggered in the dynamic analysis. Subsequently, we employed the attribution results from our dynamic analysis to match with the class names in the static analysis, identifying vulnerable third-party code. Specifically, our analysis revealed that 1634/4121 (39.7%) Chinese apps and 1094/5452 (20.1%) Google Play apps contain calls to the default certificate validation override functions. Furthermore, we identified 684/1634 (41.9%) Chinese apps and 54/684 (7.9%) Google Play apps that utilize a third-party library that we identified as insecure from our dynamic analysis. Consequently, through a combination of dynamic and static analyses, we observed 1937/4121 (47.0%) Chinese apps and 1162/5452 (21.3%) Google Play apps utilizing these override functions, with 927/1937 (47.9%) Chinese apps and 57/1162 (4.9%) Google Play apps incorporating insecure validation functions through their third-party libraries; see Table 4 for the number of occurrences of popular insecure libraries among the apps through both static analysis and dynamic analysis with their intersection.

SDK Name	SDK Package	$\mid D \mid$	$\mid S \mid$	$\mid D \cap S \mid$	$\mid D \cup S \mid$
Tencent Bugly	com.tencent.bugly	342	444	138	684
Baidu Location	com.baidu.location	39	107	17	129
Bytedance	com.bytedance.sdk	32	251	7	276
JingDong Kelper	com.kepler.sdk	14	27	7	34
Kuaishou	com.kuaishou.weapon	9	21	4	26
Alibaba Log	com.alibaba.mtl	5	1	0	6

Table 4: Example high-profile libraries that hijack certificate validation; D represents libraries found in dynamic analysis, S represents libraries found in static analysis; $D \cap S$ denotes libraries observed in both dynamic and static analyses, while $D \cup S$ denotes libraries observed in either dynamic or static analysis.

Note that we only use the static analysis to reflect the prevalence of the insecure override issue from a different angle, and the presence of a certificate validation function override in the static analysis may not mean it is actually used/called, e.g., dead code that is never reached at runtime.

Root cause. The behavior of HTTP implementations varies in terms of retrieving default values for `SSLSocketFactory` and `HostnameVerifier`, as we observe in various libraries like Apache HttpClient, Volley, and Square OkHttp. While these libraries typically abstain from retrieving such values, Android's OkHttp implementation notably does so post-initialization, potentially utilizing insecure validation functions. Conversely, Square OkHttp (from version 1.5, March 2014) remains immune by refraining from retrieval before an HTTPS call. As we observed, third-party libraries often prefer Android OkHttp, while first-party Android app code predominantly opts for Square OkHttp. We also identified instances where the default certificate verifier was overridden with an insecure verifier, yet Android's OkHttp continued to produce secure HTTPS requests. Further investigation revealed that this behavior is due to the use of an HTTP connection pool inherited from Square OkHttp. New HTTP/S connections in a pool continue to reuse previously available TLS connections (prior to hijacking), although they remain vulnerable and exploitable by an attacker resetting the connection to induce the library to establish a new TLS connection. Android appears to have incorporated the use of `DefaultSSLSocketFactory`, encompassing `X509TrustManager` and `DefaultHostnameVerifier`, to address issues related to the SPDY protocol [22]. Lastly, the multifaceted role of `SSLSocketFactory` in TLS customization, including cipher list modification, may further justify the utilization of such values.

It is also worth noting that validation override is not limited to Android OkHttp. Through manual code investigation, we found that other libraries (e.g., *jcabi-http* [13] and *ion* [12]) exhibit similar behaviors, although we did not encounter any calls to their code in our dataset. In short, although all the validation problems we found in our dataset are due to Android OkHttp, other HTTP client implementations with similar behavior can also cause the same issues.

Mitigation for global override. Possible solutions to mitigate the global certificate verifier override can be implemented/enforced by the Android OS (in the longer-term), and by developers (for immediate benefit). The Privacy Sandbox as introduced in Android 13, which separates the runtime environment of libraries/SDKs from that of the app, may address this issue if adopted and enforced. However, we did not observe any apps that use this feature in our work. If developers cannot use the Privacy Sandbox, possibly due to incompatibility with third-party SDKs, they can consider alternative short-term solutions. For example, they can use public key pinning, or re-assign the Android default `SSLSocketFactory` and `HostnameVerifier` for *every* `SSLSocketFactory` class that initiates a connection (i.e., to prevent the use of a hijacked verifier). Developers can also use other HTTP clients (e.g., Square OkHttp, Apache) that do not introduce the problematic override behaviors as in the default Android OkHttp.

Limitations. Our findings are a lower bound of the actual validation failures due to the following factors. (i) In the face of strong commercial packers, for a small portion of the packed apps with DBI detection, we analyze them up to when they crash after which Frida hooking is turned off, i.e., additional attribution data is unavailable. (ii) Certain DBI-aware apps may not crash but avoid misbehavior silently. (iii) Although we identified all the app validation issues on both the native and Java sides, we were unable to hook native certificate validation functions due to functions in .so libraries being not standardized, or the libraries often being obfuscated; as a result, we could not collect attribution data from such native functions. Furthermore, our determination of third-party code/hostnames can be further augmented with an exhaustive list (accurate but not scalable, cf. Muslukhov et al. [32]).

8 Conclusion

We shed light on the significant threat posed by improper TLS certificate validation in the Android ecosystem, especially when overriding/hijacking the default validation functions is possible unbeknownst to the parties involved in an app's operation. We conducted an automated analysis and attributed the improper validation to specific packages in an app, as opposed to the entire app. We revealed that third-party libraries play a more critical role in improper TLS validation compared to app code, which developers may be unable to address easily. The root cause as we identified is not however these third-party developers, but Google's modifications to Android's default OkHttp implementation, which Android security team, as of writing did not take any steps to rectify. Our findings emphasize the urgent need for increased awareness among app developers regarding TLS security.

References

[1] 360 Mobile Assistant - Qihoo Appstore. http://zhushou.360.cn/.

[2] 360 reinforced guarantee - 360 security. https://jiagu.360.cn/#/global/index.

[3] Aliju security - application hardening. https://jaq-doc.alibaba.com/docs/doc.htm?treeId=243&articleId=105508&docType=1.

[4] APKPure - Download APK on Android with free online APK downloader. https://apkpure.com/.

[5] Baidu Location SDK. https://lbsyun.baidu.com/products/location.

[6] Bangcle Security. https://www.bangcle.com/.

[7] ByteDance SDK. https://github.com/bytedance/tiktok-business-android-sdk.

[8] eadb - eBPF Android Debug Bridge. https://github.com/tiann/eadb.

[9] eBPF - Extended Berkeley Packet Filter. https://ebpf.io/.

[10] Facebook SDK for Android. https://developers.facebook.com/docs/android/.

[11] iJiami: Sharing IoE and guarding the smart world. https://www.ijiami.cn/enindex.

[12] Ion - Android asynchronous networking and image loading. https://github.com/koush/ion.

[13] Jcabi-http - fluent java HTTP client. https://square.github.io/okhttp/.

[14] Magisk. https://github.com/topjohnwu/Magisk/releases.

[15] Mitmproxy. https://mitmproxy.org/.

[16] NetEase Yidun - one-stop security solution. https://dun.163.com/locale/en.

[17] Tencent Bugly SDK. https://bugly.qq.com/docs/.

[18] The AppInChina App Store Index: the market-leading index of China's largest Android app stores. https://appinchina.co/market/app-stores/.

[19] Twitter4j SDK. https://twitter4j.org/.

[20] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, Valerio Persico, and Antonio Pescapé. Mirage: Mobile-app traffic capture and ground-truth creation. In *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pages 1–8, 2019.

[21] Androguard. Androguard, 2024. https://github.com/androguard/androguard.

[22] Android, Android Source Code. https://cs.android.com/android/platform/superproject/main/+/main:external/okhttp/repackaged/android/src/main/java/com/android/okhttp/HttpsHandler.java;l=101-102;drc=83f1f55b26800dfa1e5472dd5a42f598f4e3c224 Accessed 19-02-2024.

[23] Stefano Berlato and Mariano Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in Android apps. *Journal of Information Security and Applications*, 52, 2020.

[24] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following Devil's Footprints: Cross-platform analysis of potentially harmful

libraries on Android and iOS. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2016.

[25] Huajun Cui, Guozhu Meng, Yan Zhang, Weiping Wang, Dali Zhu, Ting Su, Xiaodong Zhang, and Yuejun Li. TraceDroid: A robust network traffic analysis framework for privacy leakage in Android apps. In *Science of Cyber Security*, pages 541–556, Matsue, Japan, 2022.

[26] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. Free for all! Assessing user data exposure to advertising libraries on Android. In *NDSS*, San Diego, CA, USA, February 2016.

[27] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[28] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about Android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[29] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 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*, pages 50–61, 2012.

[30] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60, 2013.

[31] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49, 2012.

[32] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic API misuse in Android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 133–146, 2018.

[33] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share first, ask later (or never?)-Studying violations of GDPR's explicit consent in Android apps. In *Proceedings of the USENIX Security Symposium (USENIX Security'21)*, 2021.

[34] Ole André Vadla Ravnås. Frida, 2022. https://frida.re/.

[35] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory still love Android: Revisiting TLS (In)Security in Android applications. In *Proceedings of the USENIX Security Symposium (USENIX Security'21)*, pages 4347–4364, August 2021.

[36] Andrea Possemato and Yanick Fratantonio. Towards HTTPS everywhere on Android: We are not there yet. In *Proceedings of the USENIX Security Symposium (USENIX Security'20)*, pages 343–360, 2020.

[37] Sajjad Pourali, Nayanamana Samarasinghe, and Mohammad Mannan. Hidden in plain sight: Exploring encrypted channels in Android apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2445–2458, 2022.

[38] Amogh Pradeep, Muhammad Talha Paracha, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. A comparative analysis of certificate pinning in Android & iOS. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 605–618, 2022.

[39] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 2455–2472, 2019.

[40] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362, 2017.

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

[42] Joel Reardon, Nathan Good, Robert Richter, Narseo Vallina-Rodriguez, Serge Egel-man, and Quentin Palfrey. Jpush away your privacy: A case study of jiguang's android SDK. *International Computer Science Institute*, 2020.

[43] RFC 5280. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, 2008. https://tools.ietf.org/html/rfc5280.

[44] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. Privacy leak identification in third-party Android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–6. IEEE, 2022.

[45] Anastasia Shuba and Athina Markopoulou. NoMoATS: Towards automatic detection of mobile tracking. *Proceedings on Privacy Enhancing Technologies*, 2020(2), 2020.

[46] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and L. Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Network and Distributed System Security Symposium*, 2014.

[47] StatCounter. Mobile operating system market share worldwide, 2022. `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

[48] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond Google Play: A large-scale comparative study of Chinese Android app markets. In *Proceedings of the Internet Measurement Conference 2018*, pages 293–307, 2018.

[49] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, JinWei Dong, Nicolas Serrano, Haoran Lu, XiaoFeng Wang, et al. Understanding malicious cross-library data harvesting on Android. In *Proceedings of the USENIX Security Symposium (USENIX Security'21)*, pages 4133–4150, 2021.

[50] Yingjie Wang, Xing Liu, Weixuan Mao, and Wei Wang. DCDroid: Automated detection of SSL/TLS certificate verification vulnerabilities in Android apps. In *Proceedings of the ACM Turing Celebration Conference-China*, pages 1–9, 2019.

[51] Yingjie Wang, Guangquan Xu, Xing Liu, Weixuan Mao, Chengxiang Si, Witold Pedrycz, and Wei Wang. Identifying vulnerabilities of SSL/TLS certificate verification in Android apps with static and dynamic analysis. *Journal of Systems and Software*, 167:110609, 2020.

[52] Ka Lok Wu, Man Hong Hue, Ngai Man Poon, Kin Man Leung, Wai Yin Po, Kin Ting Wong, Sze Ho Hui, and Sze Yiu Chau. Back to school: On the (in)security of academic VPNs. In *Proceedings of the USENIX Security Symposium (USENIX Security'23)*, Anaheim, CA, USA, August 2023.

[53] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. Happer: Unpacking Android apps via a hardware-assisted approach. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1641–1658. IEEE, 2021.

[54] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed Android malware. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings*, pages 359–381. Springer, 2015.

[55] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: Toward extracting hidden code from packed Android applications. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*, pages 293–311. Springer, 2015.

[56] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. NCScope: hardware-assisted analyzer for native code in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 629–641, 2022.

Appendix

Effectiveness of eBPF Traffic Redirection

To demonstrate the prevalence of such proxy evasion apps and confirm the effectiveness of introducing eBPF for traffic redirection, we conducted experiments on both Chinese and Google Play apps. We evaluate the eBPF redirection effectiveness from the following aspects: 1) How many apps do not respect the system proxy? 2) How many HTTPS connections are redirected to our proxy? We found that 1979/2765 (71.6%) Chinese apps had at least one connection in either HTTP or HTTPS protocols that did not respect system proxy (i.e., these apps will not be fully analyzed by existing work [33,44] , and 1928/2765 (69.7%) had their HTTPS traffic not utilized the system proxy. Our eBPF program redirected a total of 103952 HTTPS connections to our proxy. In Google Play apps, we noticed that 3394 (67.1%) apps had at least one connection that did not respect the system proxy in either HTTP or HTTPS protocols, and 3387 apps had at least one HTTPS connection redirected to our proxy (a total of 211838 HTTPS connections).

Certificate Type	Chinese		Google Play	
	# TLS Conn	% Insecure	# TLS Conn	% Insecure
Unverified Cert. Signature	64696	21.5%	75022	2.6%
Self-signed Cert.	65011	21.4%	67945	2.6%
Expired Cert.	62725	20.4%	65096	2.6%
Domain Mismatch	45736	18.8%	32509	3.7%

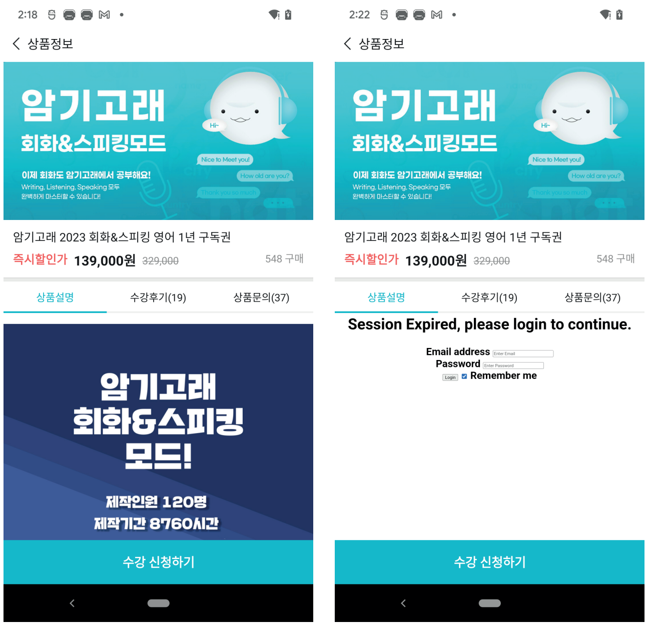
Table 5: The total TLS connections and the percentage of insecure ones

App	Category	#DL	Hijacked	Affected Party	Affected Functionality	Vulnerability	Responsible Party
com.divoom.Divoom	Art&Design	500K+	Yes	Facebook, Twitter	OAuth Authentication	Account takeover	Tencent Bugly
com.guixue.m	Education	1M+	Yes	App	SSO Authentication	Account takeover	Tencent Bugly
ch999.app.UI	Shopping	1M+	Yes	App	SSO Authentication	Account takeover	Tencent Bugly
asia.share.superayiconsumer	Tools	975K+	Yes	App	SSO Authentication	Account takeover	Baidu Location
com.paint.bynumber. color.coloringgames	Game	100M+	Yes	AppLovin	Advertisement	Phishing attack	Tencent Bugly
com.playit.videoplayer	Leisure	100M+	Yes	Flat Ads, Moloco, Smaato Ads, AppLovin	Advertisement	Phishing attack	App
asia.share.superayiconsumer	Tools	975K+	Yes	QQ Map, China Mobile	PII, GPS	Information Leaks	Baidu Location
cn.com.jschina.news	News	1.5M+	Yes	Jpush, Getui	PII, GPS, Installed apps	Information Leaks	Tencent Bugly
com.newsweekly.livepi	News	1.4M+	Yes	Taobao	PII	Information Leaks	Alibaba SDK
com.android.tutuerge	Education	10M+	Yes	Tencent Cloud	Video Play	Remote Command Execution	Tencent Bugly
cn.yonghui.hyd	Shopping	10M+	No	App	SSO Authentication	Account takeover	App
android.jianzhilieren	Tools	5M+	No	App	SSO Authentication	Account takeover	App
cn.mopon.film.xflh	Tools	290K+	No	App	SSO Authentication	Account takeover	App
com.belugaedu.amgigorae	Education	1M+	No	App	SSO Authentication	Phishing attack	App

Table 6: Exploited Vulnerabilities Introduced by Insecure TLS Validation

Category	Data Type	Chinese Apps				Google Play Apps			
		Regular	Encrypted	Insecurely Encrypted	Overall	Regular	Encrypted	Insecurely Encrypted	Overall
Device	Device ID	94	73	13	145	25	8	3	32
	Advertising ID	267	149	4	354	64	21	14	68
	Bootloader	2	11	0	13	3	1	1	4
	Build Fingerprint	276	104	9	356	26	6	1	30
	CPU Model	204	135	0	323	16	6	1	22
	Display ID	1259	211	15	1260	174	22	13	175
	Device Name	364	217	84	516	43	21	11	57
	Device Resolution	10	41	18	50	17	5	3	22
	Device ABI	470	262	44	583	46	14	10	50
	Device Model	1327	621	156	1332	206	35	23	207
	Dummy0 Interface	17	2	1	19	1	0	0	1
Network	Operator	116	132	35	233	28	14	11	35
	Device WiFi IP	63	149	7	209	9	6	1	14
	Device WiFi IPv6	2	0	2	0	0	0	0	0
	Default Gateway IP	61	128	7	186	2	2	1	4
	WiFi MAC	0	0	0	0	0	0	0	0
Network Location	Router ESSID	9	21	0	30	1	0	0	1
	Router BSSID	9	21	0	30	0	0	0	0
	neighbor Router ESSID	8	22	2	30	0	0	0	0
	neighbor Router BSSID	8	20	0	28	0	0	0	0
GPS	GPS (≤7 meter accuracy)	5	8	1	13	2	0	0	2
	GPS (78 meter accuracy)	16	15	8	31	9	0	0	9
	GPS (787 meter accuracy)	30	15	8	45	13	0	0	13
User Assets	List of Apps	26	149	22	173	0	1	0	1
	SMS	0	0	0	0	0	0	0	0
	Phone Number	59	6	4	65	4	0	0	4
	Contacts	0	0	0	0	1	0	0	1
	Device Email	4	0	0	4	15	0	0	15
	App Authentication Password	6	0	0	6	6	0	0	6

Table 7: On-device information transmission for Chinese and Google Play apps involves an insecure, improperly verified TLS certificate, with or without custom encryption (applied on top of the TLS protocol). These statistics are listed in the Regular, Encrypted, and Insecurely Encrypted columns. The overall column represents the number of apps that utilize either regular HTTPS or custom encrypted channels. In the Encrypted columns, the security level is unknown. In the Insecurely Encrypted columns, it means the app used a symmetric algorithm with a hard-coded or fixed key, or transmitted its key on a channel encrypted by a hard-coded key or fixed key.



2:18

< 상품정보

암기고래 회화&스피킹모드

이제 회화도 암기고래에서 공부해요!

Writing, Listening, Speaking 모두 완벽하게 마스터할 수 있습니다!

암기고래 2023 회화&스피킹 영어 1년 구독권

즉시할인가 139,000원 329,000 548 구매

상품설명 수강후기(19) 상품문의(37)

암기고래 회화&스피킹 모드!

제작인원 120명

제작기간 8760시간

수강 신청하기

Before

2:22

< 상품정보

암기고래 회화&스피킹모드

이제 회화도 암기고래에서 공부해요!

Writing, Listening, Speaking 모두 완벽하게 마스터할 수 있습니다!

암기고래 2023 회화&스피킹 영어 1년 구독권

즉시할인가 139,000원 329,000 548 구매

상품설명 수강후기(19) 상품문의(37)

Session Expired, please login to continue.

Email address Enter Email

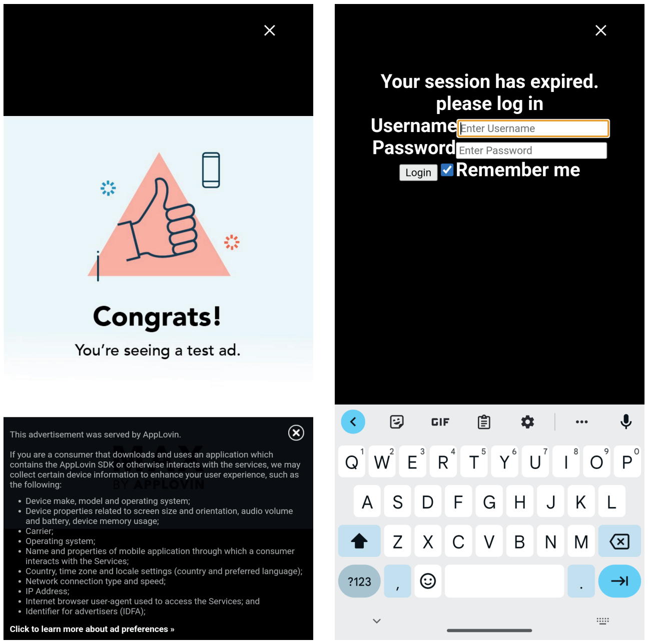
Password Enter Password

Login ☑ Remember me

수강 신청하기

After

Figure 3: Example of launching a phishing attack, by manipulating the received content from the server in the "Memorization Whale" app (`com.belugaedu.amgigorae`).



Congrats!

You're seeing a test ad.

This advertisement was served by AppLovin.

If you are a consumer that downloads and uses an application which contains the AppLovin SDK or otherwise interacts with the services, we may collect certain device information to enhance your user experience, such as the following:

- Device make, model and operating system;
- Device properties related to screen size and orientation, audio volume and battery, device memory usage;
- Carrier;
- Operating system;
- Name and properties of mobile application through which a consumer interacts with the Services;
- Country, time zone and locale settings (country and preferred language);
- Network connection type and speed;
- IP Address;
- Internet browser user-agent used to access the Services; and
- Identifier for advertisers (IDFA);

Click to learn more about ad preferences »

Before

Your session has expired. please log in

Username Enter Username

Password Enter Password

Login ☑ Remember me

After

Figure 4: Example of launching a phishing attack, by manipulating applovin's ad content in the "Paint by Number: Coloring Game" app.

```
    /**
     * Creates an OkHttpClient suitable for creating @link HttpsURLConnection instances on
     * Android.
     */
    // Visible for android.net.Network.
    public static OkUrlFactory createHttpsOkUrlFactory(Proxy proxy) {
        // The HTTPS OkHttpClient is an HTTP OkHttpClient with extra configuration.
        OkUrlFactory okUrlFactory = HttpHandler.createHttpOkUrlFactory(proxy);

        // All HTTPS requests are allowed.
        OkUrlFactories.setUrlFilter(okUrlFactory, null);

        OkHttpClient okHttpClient = okUrlFactory.client();

        // Only enable HTTP/1.1 (implies HTTP/1.0). Disable SPDY / HTTP/2.0.
        okHttpClient.setProtocols(HTTP_1_1_ONLY);

        okHttpClient.setConnectionSpecs(Collections.singletonList(TLS_CONNECTION_SPEC));

        // Android support certificate pinning via NetworkSecurityConfig so there is no need to
        // also expose OkHttp's mechanism. The OkHttpClient underlying https HttpsURLConnections
        // in Android should therefore always use the default certificate pinner, whose set of
        // @code hostNamesToPin is empty.
        okHttpClient.setCertificatePinner(CertificatePinner.DEFAULT);

        // OkHttp does not automatically honor the system-wide HostnameVerifier set with
        // HttpsURLConnection.setDefaultHostnameVerifier().
        okUrlFactory.client().setHostnameVerifier(HttpsURLConnection.getDefaultHostnameVerifier());
        // OkHttp does not automatically honor the system-wide SSLSocketFactory set with
        // HttpsURLConnection.setDefaultSSLSocketFactory().
        // See https://github.com/square/okhttp/issues/184 for details.
        okHttpClient.setSslSocketFactory(HttpsURLConnection.getDefaultSSLSocketFactory());

        return okUrlFactory;
    }
```

Listing 8: Android's modification to the Square OkHttp implementation that introduced the hijacking issue - *com.android.okhttp.HttpsHandler* class (the problematic lines are highlighted in red)

Certificate Type	**Chinese**		**Google Play**	
	% App Conn	**% Lib Conn**	**% App Conn**	**% Lib Conn**
Unverified Cert. Signature	23.0%	77.0%	85.3%	14.7%
Self-signed Cert.	23.2%	76.8%	82.9%	17.1%
Expired Cert.	23.7%	76.3%	83.1%	16.9%
Domain Mismatch	24.0%	76.0%	83.7%	16.3%

Table 8: The proportion of insecure TLS connections originating from apps and libraries