Towards HTTPS Everywhere on Android: We Are Not There Yet

Andrea Possemato
IDEMIA and EURECOM

Yanick Fratantonio
EURECOM

Abstract

Nowadays, virtually all mobile apps rely on communicating with a network backend. Given the sensitive nature of the data exchanged between apps and their backends, securing these network communications is of growing importance. In recent years, Google has developed a number of security mechanisms for Android apps, ranging from multiple KeyStores to the recent introduction of the new Network Security Policy, an XML-based configuration file that allows apps to define their network security posture.

In this paper, we perform the first comprehensive study on these new network defense mechanisms. In particular, we present them in detail, we discuss the attacks they are defending from, and the relevant threat models. We then discuss the first large-scale analysis on this aspect. During June and July 2019, we crawled 125,419 applications and we found how only 16,332 apps adopt this new security feature. We then focus on these apps, and we uncover how developers adopt weak and potentially vulnerable network security configurations. We note that, in November 2019, Google then made the default policy stricter, which would help the adoption. We thus opted to re-crawl the same dataset (from April to June 2020) and we repeated the experiments: while more apps do adopt this new security mechanism, a significant portion of them still do not take fully advantage of it (e.g., by allowing usage of insecure protocols).

We then set out to explore the root cause of these weaknesses (i.e., the *why*). Our analysis showed that app developers often copy-paste vulnerable policies from popular developer websites (e.g., StackOverflow). We also found that several popular ad libraries *require* apps to weaken their security policy, the key problem lying in the vast complexity of the ad ecosystem. As a last contribution, we propose a new extension of the Network Security Policy, so to allow app developers to embed problematic ad libraries without the need to weaken the security of their entire app.

1 Introduction

Nowadays, users rely on smartphones for a variety of security-sensitive tasks, ranging from mobile payments to private communications. Virtually all non-trivial mobile apps rely on communication with a network backend. Given the sensitive nature of the data exchanged between the app and the backend, developers strive to protect the network communication by using encryption, so that network attackers cannot eavesdrop (or modify) the communication content. However, several works have shown how properly securing network connections is still a daunting challenge for app developers.

Within the context of Android, in recent years, Google has introduced several new network security features to tackle these problems. For example, starting from Android 4.x, Android started to display alert information to the user if a "custom" certificate was added to the set of trusted CAs. Later versions of Android started supporting two different repositories for CAs: the *System KeyStore*, which contains the "default" set of trusted CAs; and the *User KeyStore*, which contains custom CAs "manually" added by the user. This separation allows Google to make apps trust only the system CAs by default. From Android 6.0, Google started to push towards "HTTPS everywhere" even further. It first introduced a new app attribute (that could be specified in the app's manifest) to specify whether cleartext (HTTP) connections should be allowed or blocked. It then extended these settings by introducing the *Network Security Policy* (NSP, in short): this mechanism allows a developer to specify complex policies (with an XML configuration file) affecting the network security of her app.

Motivated by these recent changes and by their potential security impact on the ecosystem, in this paper we present the first comprehensive study on these new defense mechanisms. We first discuss in detail these new features, the attacks that are mitigated by the NSP, and the relevant threat models. We then highlight several security pitfalls: since the policy allows the developer to define very complex configurations, they are prone to misconfigurations. We identified several patterns for which policies may provide a false sense of security, while, in fact, they are not useful.

Guided by these insights, we then present the first analysis of the adoption of the Network Security Policy on the Android ecosystem. This analysis, performed over 125,419 Android apps crawled during June and July 2019, aims at characterizing how developers are using these new features and whether they are affected by misconfigurations. The results are concerning. We found that only 16,332 apps are defining a NSP and that more than 97% of them define a NSP to allow cleartext protocols. Since starting from November 2019 Google changed some important default values related to NSP (and especially related to cleartext), we repeated the experiments over a fresh crawl of the same dataset (performed from April to June 2020): Our results show that while more apps do adopt this new security

mechanism, a significant portion of them still do not take fully advantage of it (e.g., by allowing usage of insecure protocols).

We then set out to explore *why* apps adopt such permissive policies. We found that many of these policies are simply copy-pasted from popular developer websites (e.g., StackOverflow). Upon closer inspection, we also found how many of the weak policies could be "caused" by embedding advertisement libraries. In particular, we found that the documentation of several prominent ad libraries *requires* app developers to adapt their policy and make it very permissive, for example by allowing the usage of cleartext within the entire application. While the NSP format provides a mechanism to indicate a domain name-specific policy, we found that the complex ad ecosystem and the many actors that are part of it make it currently impossible to adopt safer security policies. Thus, as another contribution of this paper, we designed and implemented an extension of the current Network Security Policy, which allows developers to specify policies at the "app package" granularity level. We then show how this proposal enables app developers to embed ad libraries without the need of weakening the policy of the core app, how it is fully backward compatible, and how it can thus act as a drop-in replacement of the current version.

In summary, this paper makes the following contributions:

- We perform the first comprehensive study on the newly introduced Android network security mechanisms, identifying strengths and common pitfalls.

- We perform the first large-scale analysis on the adoption of the Network Security Policy on the Android ecosystem, using a dataset of 125,419 apps. Our study found that a significant portion of apps using the NSP are still allowing cleartext.

- We investigate the root causes leading to weak policies, and we found that several popular ad libraries and the complex advertisement ecosystem encourage unsafe practices.

- We propose a drop-in extension to the current Network Security Policy format that allows developers to comply with the needs of third-party libraries without weakening the security of the entire application.

In the spirit of open science, we release all the source code developed for this paper and the relevant datasets.

2 Network Communication Insecurity

This section explores the different threats that an application might be exposed to due to insecure network communications. For each of the issues, we also discuss the relevant threat models.

2.1 HTTP

An application using a cleartext protocol to exchange data with a remote server allows an attacker to mount so-called Man-In-The-Middle (MITM) attack, through which it is possible to eavesdrop (or even modify) the network traffic at will. This, in turn, can lead to the compromisation of the user's private information or of the application itself [4, 32, 37, 49]. The actual severity of this threat changes depending on the nature of the data exchanged by the application and the network backend. In other words, this HTTP scenario can be exploited by an attacker within the following threat model:

Threat Model 1. An attacker on the same WiFi network (or on the network path) of the victim can eavesdrop and arbitrarily modify apps' unencrypted connections and data at will.

2.2 HTTPS and Certificate Pinning

By adopting the "secure" version of HTTP, *HTTPS*, it is possible to perform network operations over a secure and encrypted channel. Exchanging data using HTTPS (SSL/TLS) ensures integrity, confidentiality, and authenticity over the connection between the application and the remote server. This mechanism works as follows. First, when an application tries to contact a remote server using SSL/TLS, a "handshake" is performed. During this phase, the server first sends its certificate to the client. This certificate contains multiple pieces of information including its domain name and a cryptographic signature by a so-called Certificate Authority (CA). To determine whether the client should trust this CA, the system consults a set of hardcoded public keys of the most important (and trusted) CAs: If the certificate is signed (directly or indirectly) by one of these CAs, the certificate is then considered trusted and the (now secure) connection can proceed; otherwise, the connection is interrupted [1].

While SSL/TLS is a powerful mechanism, it can be compromised by an attacker within the following threat model:

Threat Model 2. An attacker that can obtain a rogue certificate can perform MITM over HTTPS connections. We consider a certificate to be "rogue" when it is correctly signed by a (compromised) trusted CA without an attacker owning the target domain name [2, 33].

Attacks within this threat model can be mitigated by implementing *Certificate Pinning*. Certificate pinning consists in "hardcoding" (or, pinning) which is the expected certificate(s) when performing a TLS handshake with a given server. From the technical standpoint, this "expectation" is hardcoded within the application itself, and the app can thus verify, during the handshake, that the certificate sent from the server matches with the expected one. Even though pinning is a powerful security mechanism, previous works have shown how it is very challenging to properly implement it. In fact, to implement pinning, developers are tasked to rely on a wide variety of libraries, each of which exposes a distinct set of APIs. Handling diverse implementations of pinning may push developers to take some shortcuts: It was shown how it is not uncommon for developers to rely on "ready-to-use," but *broken*, implementations of certificate pinning copied from websites like StackOverflow [25].

These broken implementations might lead to accepting arbitrary certificates without even verifying which CA signed them, or whether the certificate was issued for the given domain. Moreover, it has also been shown how even popular network libraries themselves may fail to properly implement pinning [31].

2.3 User Certificates

The Android system comes with a set of pre-installed CAs to trust and uses them to determine whether a given certificate should be trusted. These CAs reside in a component named *KeyStore*. The system also allows the user to specify a *User KeyStore* and to install custom CAs. There might be situations where the custom CAs allow to perform a MITM over SSL/TLS connections (see Section 4). However, performing MITM over a secure connection should not always be considered a malicious activity. For example, proxies used to debug network issues rely on the same technique. Self-signed certificates generated by these tools do not have a valid trust chain and thus cannot be verified, and the app would terminate the connection. By adding a custom CA, apps can successfully establish a network connection.

Unfortunately, *User KeyStore* and self-signed certificates can also be abused by malware. Of particular importance is the emerging threat of "stalkware" (also known as "spouseware") [17, 29]. In short, this scenario can be exploited by an attacker within the following threat model:

Threat Model 3. An attacker that has physical access to the device can silently install a new custom certificate to the *User KeyStore*, and mount MITM (including on HTTPS connections) to spy the user's activities.

3 Network Security Policy

To make the adoption and implementation of "secure connections" easier for a developer, Google recently introduced several modifications and improvements, which we discuss in this section.

The first problem that Google tried to address relates to the installation of *self-signed certificates*. In very early versions of Android, it was possible to silently install one of these certificates, thus allowing anyone who controls it to perform MITM on SSL/TLS connections. In Android 4.4, however, Google introduced the following change: if a self-signed certificate is added to the device, the system would display a warning message informing the user about the risks and consequences of MITM on SSL traffic [45]. However, since there might be scenarios where trusting a (benign) self-signed certificate is necessary (e.g., to perform network debugging), Google decided to split the KeyStore into two entities. The first one, named *System KeyStore*, is populated with pre-installed CAs, while the second one, named *User KeyStore*, allows the user to install self-signed certificates without altering the System KeyStore.

The second problem Google tried to mitigate is the adoption of *cleartext protocols* [3]. Starting from Android 6.0, Google introduced a new security mechanism to help apps preventing cleartext communication, named *Network Security Policy* [5]. With this new policy, an app can specify the `usesCleartext-Traffic` boolean attribute in its manifest file and, by setting it to false, the app can completely opt-out from using cleartext protocols, such as HTTP, FTP, IMAP, SMTP, WebSockets or XMPP [7]. Moreover, from Android 7.0, the new default is that apps do not trust CAs added to the User KeyStore [14]. It is possible to override this default, but the developer needs to explicitly specify the intention of using the *User CAs* within the policy.

Note that, from an implementation point of view, the policy is *not* enforced by the operating system (as it would be impractical), but it is up to the various network libraries to actually honor it (e.g., by interrupting an outbound HTTP connection if cleartext traffic should not be allowed). Note also that, to address backward compatibility concerns, for an app targeting an API level from 23 to 27 (i.e., from Android 6.0 to Android 8.1), the default value of the `usesCleartextTraffic` attribute is `true`. However, if an app targets API level 28 or higher (i.e., Android 9.0+), then the default for that attribute is `false`, forcing developers to explicitly opt-out from this new policy in case their apps require HTTP traffic.

While this policy is a significant improvement, for some apps it may currently be impractical to completely opt-out from cleartext communications. In fact, this policy follows an "all-or-nothing" approach, which might be too coarse-grained. This is especially true when a developer is not in complete control of its codebase, such as when embedding closed-source third-party libraries. In fact, these third-party libraries may reach out to remote servers using cleartext protocols or to some domain names that are not even supporting HTTPS. To allow for a more granular specification, with the release of Android 7.0, Google introduced an extended version of the NSP, which we discuss next.

3.1 Policy Specification

The new version of the NSP, introduced by Google in Android 7.0, has undergone a complete redesign [6]. The policy now resides on an external XML file and it is not mixed anymore with the AndroidManifest. The most interesting feature introduced in this new version is the possibility to specify additional network security settings other than allowing or blocking cleartext protocols. Moreover, to overcome the lack of granularity of the previous version, the policy now allows for more customizations through the introduction of the new `base-config` and `domain-config` XML nodes. The semantics of these two nodes is the following: all the security settings defined within the `base-config` node are applied to the entire application (i.e., it acts as a sort of default); the `domain-config` node, instead, allows a developer to explicitly specify a list of domains for which she can specify a different policy.

Cleartext. Allowing or blocking cleartext protocols can now be easily achieved with the `cleartextTrafficPermitted` attribute. Moreover, the developer can decide "where" to apply

this security configuration. This attribute can be defined both within a "base" and "domain" config node. To enforce this setting at runtime, networking libraries can rely on the `Network-SecurityPolicy.isCleartextTrafficPermitted()` API, which returns whether cleartext traffic should be allowed for the entire application. Instead, to check if cleartext traffic is allowed for a given host, a library can use the `isCleartextTrafficPermitted(String host)` API.

Certificate Pinning. Configuring certificate pinning is now much simpler than it was in the past. First, since certificate pinning is used to verify the identity of a specific domain, all the configurations need to be defined in a `domain-config`. Second, the developer needs to define a `pin-set` node (with an optional `expiration` attribute to specify an expiration date for this entry). The `pin-set` node works as a wrapper for one or multiple `pin` nodes, each of which can contain a base64-encoded SHA-256 of a specific server's certificate. Multiple `pin`s can be used as a form of *backup*, to avoid issues while performing *key rotations*, or to *pin* additional entities like the *Root CA* that emitted the certificate for the domain. The connection is allowed *if and only if* the hash of the certificate provided by the server matches with at least one hash in the `pin-set` node.

KeyStore and CAs. The new version of the policy allows a developer to specify which KeyStore to consider as trusted when performing secure connections. The developer has first to define a `trust-anchors` node, which acts as a container for one or more `certificate` nodes. Each `certificate` node *must* have a `src` attribute, which indicates *which* certificate(s) to trust. The values for `src` can be one of the following: `system`, which indicates that the System KeyStore, the default one; `user`, which indicates the *user-installed certificates* within the *User KeyStore*; or a path to an *X.509 certificate* within the app package. When multiple `certificate` nodes are defined, the system will trust their union.

Besides, the developer can also specify an `overridePins` boolean attribute within a `certificate` node. This attribute specifies whether the CAs within this certificate node should bypass certificate pinning. For example, if the attribute's value is `true` for the *system CAs*, then pinning is not performed on certificate chains signed by one of these CAs.

Debug. Applications protected by the NSP are more difficult to debug. To address these concerns, the policy can contain a `debug-overrides` node to indicate which policy should be enforced when the app is compiled in *debug mode*.[1] If the developer leaves a `debug-override` node in the policy of a release build, the content of the node is simply ignored.

3.2 Towards HTTPS Everywhere

Starting from Android 7.0, at apps' installation time, the system checks whether the developer did define a policy: if yes, it loads

---

[1]Apps can be compiled in *release* or *debug* mode. This can be done by setting the `android:debuggable` manifest attribute accordingly. Apps *must* be compiled in *release* mode to be accepted on the Play Store.

the policy; otherwise, it applies a default one. Note also that if a policy is defined but it does not specify a node or an attribute, the system *fills* the missing values by inheriting them from a similar node, or, when none are available, from the default configuration. The default values applied by the system do change over time depending on the target API level and are becoming stricter—and by forcing app developers to target high API levels to be admitted on the official Play Store, Google is leading a push towards *HTTPS everywhere*. We now discuss how these default values change depending on the target API level.

API 23 and Lower. An application targeting an API level *lower or equal than 23* cannot specify a policy since this mechanism was introduced from API level 24. In this case, the system will then enforce the following default policy:

```
<base-config cleartextTrafficPermitted="true">
  <trust-anchors>
    <certificates src="system" />
    <certificates src="user" />
  </trust-anchors>
</base-config>
```

This configuration allows an app to use cleartext protocols and to trust the union of CAs from both *System* and *User KeyStore*.

From API 24 to 27. The default policy for applications targeting API levels *from 24 to 27* changes as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

That is, cleartext traffic is still allowed, however, only CAs in the *System KeyStore* are trusted by the application.

API Level 28 and Higher. For apps targeting an API level *greater or equal of 28*, the policy is even stricter:

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

This change enforces that all cleartext protocols are blocked [8].

Starting from November 1st, 2019, all applications (and updates as well) published on the official Google Play Store *must target at least API level 28*, corresponding to Android 9.0 [28]. In Appendix, we report a concrete example of a (complex) policy that touches on the various points previously discussed.

3.3 TrustKit

One library that is particularly relevant for our discussion is *TrustKit* [19]. This library allows the definition of a NSP for apps targeting versions of Android earlier than 7.0 (which, as we mentioned before, do not support NSP). From a technical standpoint, this library reimplements the logic behind the NSP, allowing an application to import it as an external library. Note that TrustKit

only supports a subset of features: the developer cannot specify a `trust-anchors` within a `domain-config` node, and it is not possible to trust CAs in the User KeyStore. However, the library implements a mechanism to send *failure reports* when pinning failures occur on specific domains, allowing a developer to constantly monitor for pinning violations. Interestingly, this feature is not available by the system-implemented NSP.

4   Policy Weaknesses

As discussed in the previous section, NSP is undoubtedly making the specification of a fine-grained network policy more practical. However, each of the features introduced by the NSP may be *inadvertently disabled or weakened* by an inexperienced developer during the definition of the policy. Unfortunately, to date, there are no tools that help developers to verify the correctness of the defined policy and to check that the settings she wanted to implement are effectively the ones enforced by the system.

This section discusses several potential pitfalls that may occur when an inexperienced developer configures a NSP.

Allow Cleartext. As described in the previous section, a developer has multiple ways to define the usage of cleartext protocols. For example, the developer can define a list of domains and limit the adoption of cleartext only to them. Otherwise, if the application contacts all the endpoints securely, she can completely opt-out from cleartext communications and be sure to identify potential regression issues. However, a developer may configure her application with the following policy:

```
<base-config cleartextTrafficPermitted="true">
    ...
</base-config>
```

This configuration allows the application to use cleartext protocols, *potentially* exposing the user and the application to threats described in Section 2. To make things worse, as we will discuss throughout the paper, several online resources suggest implementing this very coarse-grained policy, with the goal of disabling the safer defaults: the main concern is whether the inexperienced developer is fully aware of the security repercussions of such policy.

For the sake of clarity, it is important to mention how this specific configuration does not impact an application where all the endpoints are already reached securely—this policy is useful only when acting as a safety net. In other words, this configuration does not lower nor weaken the security of an application performing all the network operations using, for example, HTTPS. However, this configuration is not able to identify *regression* issues: if an endpoint is inadvertently moved from HTTPS to HTTP, the insecure connection is allowed due to this "too open" policy (while the default policy could have blocked that). A similar scenario also affects complex apps, which are either developed by different teams within the same organization or that are developed by embedding a high number of third-party dependencies: in these cases, it is extremely challenging, if not out-

right impossible, to make sure that no connection would rely on cleartext protocols. Unfortunately, as we previously discussed, even one single endpoint (or resource) reached through HTTP might be enough to compromise the security of the entire app.

Certificate Pinning Override. The NSP makes the adoption and configuration of certificate pinning straightforward. The developer now only needs to declare a valid certificate for each of the domains she wants to protect: then, the system takes care of all the logic to handle the verification of the certificates at connection time. On the other hand, we identified pitfalls that an inexperienced developer may not be aware of. For example, consider the following policy (which we took from a real app):

```
<domain-config>
  <domain>DOMAIN</domain>
    <pin-set>
      <pin digest="SHA-256">VALID_HASH</pin>
    </pin-set>
  </domain-config>
<trust-anchors>
  <certificates src="system" overridePins="true"/>
</trust-anchors>
```

We argue that this policy is misconfigured and that it is very likely that the developer is not aware of it. Given the specification of the `pin-set` entries, it is clear that the intent of the developer was to actually implement certificate pinning. However, the `overridePins` attribute of the system certificate entry is set to `true`: this indicates that certificate pinning should *not* be enforced for *any* CAs belonging to the *System KeyStore*, thus making the previous `pin-set` specifications useless. We believe that this kind of policy offers a "false sense" of security for a developer, especially since no warnings are raised at compilation time nor at runtime.

Silent Man-In-The-Middle. Switching from HTTP to HTTPS does not always guarantee that the communication cannot be eavesdropped. As described in Section 2, under certain specific circumstances, it is possible to perform MITM over SSL/TLS encrypted connection and break the confidentiality, integrity, and authenticity of the communication. Consider the following policy taken from a real app:

```
<trust-anchors>
  <certificates src="system"/>
  <certificates src="user"/>
</trust-anchors>
```

This policy may expose an application to MITM (see Threat Model 3). In fact, this policy trusts the union of the CAs in the *System and User KeyStore*: hence, the traffic of the app can be eavesdropped by anyone who controls a custom CA in one of the KeyStores. This policy overrides the default configuration introduced on Android 7.0, which prevents applications from trusting CAs stored in the *User KeyStore* when performing secure connections. Even though trusting "user" certificates may be the norm at the development phase, we believe that a "production app" that actually trusts user certificate is often a symptom of misconfiguration since it is very rare that an

app would actually need to trust User CAs. For example, even network-related apps such as VPN apps do not need to trust User CAs, even when trusting custom certificates is required: in fact, VPN apps can hardcode the custom CA within the app, and add a `trust-anchors` node pointing to it. This has the net effect of trusting *only* this specific certificate, and nothing else. One scenario where trusting User CAs seems required relates to Mobile Device Management apps (MDM), which need to install different CAs coming from different sources and that cannot be pre-packaged within the released app. However, these MDM apps constitute a rare exception, rather than the norm.

5   Policy Adoption

As one of the contributions of this paper, we set out to explore how the NSP has been adopted by the Android ecosystem. This section discusses our findings, and it is organized as follows. First, we present the dataset we used for our study (§5.1). Second, we discuss how apps use this new security mechanism, we provide statistics on how frequently each feature of the policy is used, and we present insights related to apps adopting policies that are inherently "weak" and that likely constitute inadvertent misconfigurations (§5.2). Last, we conclude this section with an analysis of *network libraries*, which, from a technical standpoint, is where the "enforcing" of the policies actually lies; we have also developed an automatic testing framework to determine whether a given network library correctly honors the various elements of network policies (§5.3).

5.1   Dataset

To perform our analysis, we first built a comprehensive and representative dataset of apps. To determine which apps to download, we obtained the package names from AndroidRank [9], a service that provides "history data and list of applications on Google Play." We opted to select the "most-installed applications" on the Google Play Store according to the *installation distribution*, with apps whose unique installation count ranges from 10K to more than a billion. In total, we downloaded 125,419 apps, during June and July 2019.

5.2   Dataset Exploration & Weaknesses

Methodology. After extracting the policies from the apps, we first perform clustering to highlight common patterns and whether two or more apps share the same exact policy (or specific portions of it). In particular, we group two policies in the same cluster if they contain the same nodes, attributes, and values, in any order. This approach also helps us to determine whether apps developers "copied" policies from known developer websites, such as StackOverflow. We then analyze the clusters to identify peculiar configurations or weaknesses. Once an interesting configuration has been identified, we then proceed by performing queries on the entire dataset (that is,

inter-cluster) to measure how common this specific aspect of the configuration is and whether it affects many apps.

We then performed an additional analysis step, which is based on similar clustering techniques, but performed over a *normalized dataset*. We refer to a policy as "normalized" after we remove artifacts that are clearly specific to an app. We replace all the concrete values of domains with the value URL, all certificate hashes with HASH, and all the `expiration` dates with DATE. The rationale behind this normalization step is to be able to group policies "by semantics," which is not affected when some specific concrete values differ.

Overview. One of the first insights is that, even though the NSP was firstly introduced in Android 6.0 in 2015, we note how 109,087 of the apps do *not* implement any policy (in either of the two forms). Of the remaining 16,332 apps that do implement a policy, 7,605 of them (6% of the total) adopt the original version of the policy (available in Android 6.0), while 8,727 (6.95%) adopt the new, more expressive policy format (available in Android 7.0). Our dataset is distributed as follows: 0.5% of the apps (83) target API level 29, 75% (12,261) API level 28, 11% (1,803) API level 27, 12% (2,077) API level 26, and the remaining 0.6% (108) target API level 25 or lower.

The first clustering process creates in total 271 clusters (where a cluster is formed by *at least* two apps): these clusters group 7,184 apps out of the 8,727 apps defining the policy—the remaining 1,543 policies were unique and did not fit any cluster. The clustering process on the normalized dataset, instead, generates 170 clusters, this time with only 311 applications not belonging to any group. The remainder of this section discusses several interesting insights and common patterns.

Cleartext. Among the generated clusters, one is particularly big: it is formed by 1,595 apps. All these apps share the trivial policy of "allowing cleartext globally." The exact same configuration is also used by other 2,016 apps belonging to 60 different clusters. Among the apps not belonging to any cluster, this configuration is used by 199 of them. Thus, in total, 4,174 apps of our dataset allow cleartext for the entire app. We then investigated how many apps opted out from cleartext and we found that only 156 apps block cleartext for the entire app. Then, we considered also apps using the first version of the policy since it also allows a developer to fully opt-in, or opt-out, from cleartext. Among the 7,605 apps using the first version of the policy, 97.5% (7,416) of them allow cleartext protocols, while only the 2.48% (189) opted out from them.

As previously discussed in Section 3, the cleartext attribute can also be enabled by default if an app is targeting an API level lower or equal to 27 and it does not override it. By considering also the default settings, the numbers are even more worrisome. We noticed that among the 16,332 apps with a NSP, the 84.8% of them (13,847) allow the usage of cleartext protocols. The 12.3% (1,837) of them enable cleartext due to the default configuration not being overridden. To conclude, only the 1.2% (170) opt-out from cleartext just for a specific subset of domains.
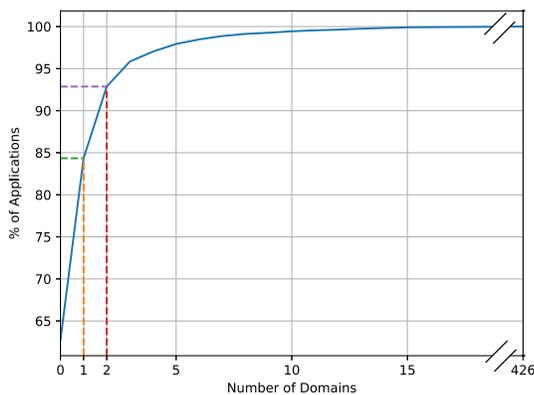
Figure 1: The figure shows the CDF of the number of domains defined within policies. Note how 62.5% of the apps do not define a custom policy for any domain. The 21% of the apps define exactly one domain, while the 8.5% specifies up to 2 domains within the policy. Note that the CDF has a long tail, with several apps defining more than 30 domains within the same policy, and two apps specifying 368 and 426 policies.

Domains. We then proceed by looking at apps using the cleartext attribute on an explicit list of domains (using the `domain` node). We identified only 2,891 apps allow cleartext for a subset of domain while only 219 force the domain in the list to be reached only securely. Figure 1 shows the cumulative distribution function (CDF) of the number of domains defined within policies. In general, most apps ($\sim 95\%$) specify custom policies for at most three domain names.

Policy for 127.0.0.1. We then looked at clusters of more complex policies, in terms of nodes and attributes, and we noticed some interesting patterns. We identify how 492 apps configure a very specific `domain-config` node for the IP address 127.0.0.1, localhost. Even if this policy does not introduce any security vulnerability and should be considered as a safe policy, we found it interesting: while it may be common practice to spawn a local server, it is very uncommon that all the 492 apps define the same policy for localhost. This configuration, however, is very common among other apps: in total, we identify other 512 apps belonging to 43 different clusters having the same `domain-config` setup, and 109 apps not belonging to any cluster. Thus, this specific domain configuration is used by 1,113 apps. We then set out to pinpoint the underlying source of this policy, and we eventually determined that this policy is defined by the *Audience Network Android SDK*, the Facebook advertisement framework. In particular, we noticed how a developer who wants to use this library *must modify* the policy to include this specific configuration to avoid unintended behavior. The official library's documentation makes clear that this modification is *mandatory* due to the internals of the library itself. This finding opens a scenario that is different than the simple "developers copy policies": in

this case, an advertisement library explicitly requested the developer to modify her policy to make the library work. We suspected that this pattern could be common to many other advertisement libraries. Unfortunately, our suspicion proved to be correct: we identified several ad libraries that explicitly request developers to copy-paste a given policy. Moreover, we found how the ad libraries' documentations often attempt to convince developers by including misleading and/or inaccurate arguments, and how many of such policies' modifications actually negatively affect the overall security of the entire app. We postpone an in-depth discussion of these findings to the next section (Section 6).

Trusted Certificates. Another interesting cluster is formed by 427 apps, which use a `trust-anchors` node for the entire app to trust the union of *System and User CAs*. As previously discussed, this configuration might allow, under specific circumstances, to perform a MITM over SSL/TLS connections (see Threat Model 3). Nonetheless, we notice how this specific configuration is shared among other 1,083 apps, 600 of which belong to 24 different clusters. We then investigate how many apps use the same configuration for a subset of domains ending up identifying 73 apps: thus, in total, we identified 1,159 apps adopting this configuration, among which 1,038 of them allow their SSL/TLS traffic to be potentially intercepted.

Domain `example.com` and Invalid Digests Another peculiar configuration comes from apps using the domain *example.com* within their policy. We identified this interesting configuration from a cluster of 41 apps. However, there is, of course, no need for an app to protect this domain since this *example.com* domain is clearly not relevant. Thus, we looked for similar apps and we found out that in total, other 58 apps use this domain, 48 of which come from 7 different clusters. We then found that these policies are copied verbatim from the Android Developer website and from StackOverflow. We tracked down the original policies combining both the domain name and the *unique* digests defined in some of the policies. These policies define certificate pinning on *example.com* or with invalid digests formed by "B" repeated 44 times (see the Appendix for the complete policies). We believe that there are two possible explanations to justify the adoption of these (useless) policies. In the first one, the developer wants to define one specific feature of the policy: she then copies an existing policy that contains both the requested feature and the unique configuration of certificate pinning. In the second one, this policy might have been used by a developer who was looking for a certificate pinning implementation and she copied the first available policy. While copying security policies that contain "dummy" domain names such as *example.com* is not a security problem per se, we believe that these policies may create a false sense of security in the developer's mind: the developer may wrongly believe that certificate pinning is correctly implemented in her application, while, in fact, it is not.

Certificate Pinning. Certificate pinning increases the security of the communication ensuring integrity, confidentiality, and authenticity. Thankfully, implementing certificate pinning via

NSP is now much simpler than it was in the past. However, we found that only 102 applications enforce it through the policy. Out of these 102 apps enforcing certificate pinning, an interesting cluster is constituted by apps that implement pinning *but then mistakenly override it*. We identified 9 apps that specify one or more `pin-set`, but set the `overridePins` attribute to `true`, making the various `pin-set` useless. We argue that it is very likely that the developer is not aware of it, otherwise she would not have specified any `pin-set` entry. We believe Android Studio (or other IDEs) should flag this kind of policy as potentially misconfigured.

Invalid Attributes. We identified a group of apps defining attributes that are not specified within the official documentation [24]. For example, we identified two apps defining the `usesClearTextTraffic` attribute in the policy (even if this is only valid in the *old* version of the NSP), or two apps defining the `cleartextTrafficPermitted` attribute within a wrong node. We also found one app declaring the `hstsEnforced` attribute, which is *not* mentioned in the official documentation. However, by looking at the source code of the policy parser, we notice how this attribute is actually recognized as valid. This attribute allows a developer to define HSTS for the WebView component of her application (which would "force" the WebView to always contact via HTTPS websites sending the HSTS header [18]). We note how the concept of HSTS significantly overlaps with the cleartext aspect of the NSP. We investigated the reason why this attribute is still available within the NSP and we found out that it may exist because older versions of the WebView were not enforcing the `cleartextTrafficPermitted` attribute [24] (but were enforcing HSTS instead).

TrustKit. The cluster of policies defined using TrustKit is formed by 53 apps. Among these apps, 10 define a `reporting-endpoint` to use when a pinning failure is identified, while 16 apps explicitly disabled this feature. To conclude, 46 apps define certificate pinning within the policy.

Remaining Apps. Our methodology based on clustering and targeted queries allowed us to systematically group a vast portion of our dataset. However, as we mentioned, 311 apps did not fit any cluster. We then manually inspected them all, to look for additional interesting patterns. Among these, we identified 98 apps that define a very unique policy in terms of `domain` nodes used with the policy. The other 46 apps shared a specific policy that did not take advantage of the "wrapper nodes" like `pin-set` or `domain-config`: for each of the domains, these apps opened a new `domain-config` node each time instead of defining all the domains within one node. We also found 44 apps that specify more than one custom certificate. Another interesting configuration comes from apps whose policy appears very verbose and that could have been reduced. We noticed how 32 applications specify a default "allow cleartext" for the entire app and, on top of that, configured a very detailed list of domains and subdomains with the same exact policy. 21 applications defined additional

text (like comments or left-over in between nodes) that is then removed by the system during the parsing process. To conclude, the remaining apps defined very unique and complex policies that do not belong to any of the aforementioned groups, but that, from the security perspective, do not represent anything special.

Dataset Evolution. Starting from November 1st, 2019, all apps *must target at least API level 28* [28]. This means, from a NSP perspective, that all the new apps, by default, will forbid cleartext. Since our dataset was crawled before November (see Section 5.1), we decided to repeat some of the measurements, this time on a dataset downloaded after this new mandatory requirement. Our goal is to investigate how the apps evolved after the introduction of the new default value that forbids the usage of any cleartext protocol. We started a re-crawl of the same initial dataset, starting from the 125,419 package names. These apps were re-crawled from April to June 2020. We were able to download 86.5% of the initial dataset, for a total of 108,542 apps. Of the remaining apps that we could not re-download, 15,749 apps were removed from the Google Play Store and 1,128 apps moved from a free to "paid" download or introduced in-app purchases not available in our geographical region. The apps that we were able to re-crawl are distributed as follows: the 14.3% of the apps (15,531) target an API level 29, the 46.2% (50,191) instead target a level 28, 9.5% (10,351) the level 27, 12.7% (13,795) level 26 and the remaining 17.2% (18,674) target an API level 25 or lower.

Unsurprisingly, the number of apps defining a NSP increased: 33.3% of the apps (36,165) now specify one of the two types of NSP. Among these apps, the 65.5% (23,718) still adopts the first version of the NSP through the `AndroidManifest`, while the remaining apps (15,492) opted for the new and more recent version. Interestingly, 8.4% of the apps (3,045) use both versions of the policies.

We then looked for how many apps effectively adopted the new default of forbidding cleartext protocols for the entire application: surprisingly, approximately the 33% of the entire dataset (35,789 out of 108,542) enforced a default configuration that does not permit cleartext protocols. Out of these apps, 419 used the first version of the policy. The remaining 67% of the apps still configure a NSP that permits cleartext traffic. From this 67%, the 32% (23,229) still adopt the first version of the policy. However, what it is interesting to notice is that 58% (42,353) of apps allow cleartext due to default configuration, dictated by the API level. To conclude, we note how only a small portion of apps, the 0.4% (349), allow cleartext as base configuration and also define a set of domains for which they allow only encrypted connections.

These results somehow highlight an ecosystem-wide problem that affects Android apps: even if Google provides a simple and easy way to configure the SSL/TLS for an app (the NSP), and even though it explicitly changed the defaults to force the usage of cleartext protocols, a significant portion of apps still opt to stick, for one reason or another, to plain and unencrypted networking protocols: while the community is making progress, we are not there yet for a full adoption of HTTPS by Android apps.

5.3 Android Networking Libraries Adoption

So far, this section has focused on the exploration of how apps adopt NSPs. However, we did not tackle the aspect of *enforcing* these policies. The NSP is simply an XML configuration file, and it is then up to the various network libraries to properly honor (and enforce) what is specified by such configuration file.

To this end, we set out to explore how Android apps and network libraries do enforce these policies. First, we checked the official Android documentation, which states that "third-party libraries are strongly *encouraged* to honor the *cleartext setting*" [24]. We found the documentation concerning, for two reasons. First, the wording of the documentation only mentions that honoring the policy is "strongly encouraged." However, we believe that since the policy relates to security-relevant aspects, network libraries should be forced to honor the policy—and in case they do not, that should be considered as a vulnerability. In fact, a network library not honoring the policy would have the negative side-effect of silently making the policy useless. Second, the documentation only mentions the "cleartext settings." However, as we discussed in Section 3, the new version of the policy touches on many more aspects: Unfortunately, the documentation does not even mention the other features (e.g., which *KeyStore* to trust, pinning).

Next, we checked the official API, implemented by the `NetworkSecurityPolicy` class. This is the API that, in theory, network libraries should rely on to obtain the content of the policy (and honor it). However, this API appears very limited: the only available API is `isCleartextTrafficPermitted()`, which returns whether cleartext traffic should be allowed. There is no other API to query the remaining fields of the policy, and it is thus not clear how network libraries are supposed to enforce them.

For these reasons, we set out to explore how and whether popular network libraries honor the policy. The remainder of this section discusses how we built a dataset of network libraries, an automatic analysis framework to test whether a given library honors the various aspects of a policy, and the results of this analysis.

Libraries Dataset. To perform this investigation, we first built a comprehensive dataset of the most used networking libraries. We identified these libraries from AppBrain [11], a service that provides multiple statistics on the Android application's ecosystem such as "Android libraries adoption" by different apps. Our dataset consists of all the network libraries mentioned by AppBrain: *URLConnection, Robospice, HttpClientAndroid, AndroidAsync, Retrofit, BasicHttpClient, OkHttp, AndroidAsyncHTTP, Volley,* and *FastAndroidNetworking*. Except for *URLConnection*, which is the default HTTP library on Android, all the libraries are "external," which means that app developers need to manually specify them as external dependencies. Note that these external libraries, even though they are not the default, are used by almost 30% of all the apps published on the Google Play Store ($\sim 250K$ unique apps). Table 3, in Appendix, provides more detailed statistics.

Analysis Framework. Determining whether a given library is implementing the NSP is not a straightforward process. In fact, the source code of these libraries is often not available, and manual reverse engineering may be challenging and error-prone. Thus, we opted for an automatic approach based on dynamic analysis. We built an automatic framework to check whether a given networking library honors the policy defined in an app. Note that while for this paper we tested the ten popular network libraries in our dataset, our framework is completely generic and can be easily used to vet an arbitrary network library.

Our framework analyzes each network library individually. For each of them, it performs the following steps. First, we generate *all* the possible combinations of a policy, by combining all possible nodes, attributes, and representative values. In particular, the framework considers the following nodes: `base-config`, `domain-config`, `pin-set`, and `trust-anchors`. For each node, it considers all the relevant child nodes, such as `domain`, `pin`, and `certificate`. Each node is then configured with all the possible attributes that might be used within a given node, like `overridePins` for what concerns `trust-anchors`, or `src` for the `certificate` node (see Section 3 for the entire list). For what concerns the values, we generate "representative values." For the value field representing a certificate hash, we generate various policies with the following values: a valid hash matching the hash of the certificate actually used during the tests, a valid hash that is different than the expected one, and a non-valid hash (e.g., the character "A" repeated several times). The combinations of all nodes, attributes, and representative values, generates 72 unique policies.

Then, the framework creates an app that attempts to connect to an endpoint via HTTP and via HTTPS by using the library under test. The app is then built multiple times, each time with a different policy. Each of these apps is then tested in three different "testing environments," each of which simulates the different threat models discussed in Section 2: 1) the app is tested without attempting to perform MITM; 2) we simulate an attacker performing MITM (by using a proxy); 3) we simulate an attacker performing MITM with the attacker's custom CA added to the User KeyStore. At each execution, the framework logs whether a given connection with a given policy in a given testing environment was successful or not. These logs are compared with a ground truth, which is generated by a Python-based implementation that takes into account the various aspects of the policy and the various testing environments. We flag a library as compliant if and only if the runtime logs match with the expectations of the ground truth.

Compliance Results. First, we identified that *HttpClientAndroid*, *AndroidAsync*, and *AndroidAsyncHTTP* are *not* enforcing the cleartext attribute: these libraries allow HTTP even though the policy would prohibit it. We note how these libraries are used by more than tens of thousands of popular apps with hundreds of millions of unique installations.

Instead, for what concerns *certificate pinning* and *trusted anchors*, we noticed that nine of the ten libraries do correctly
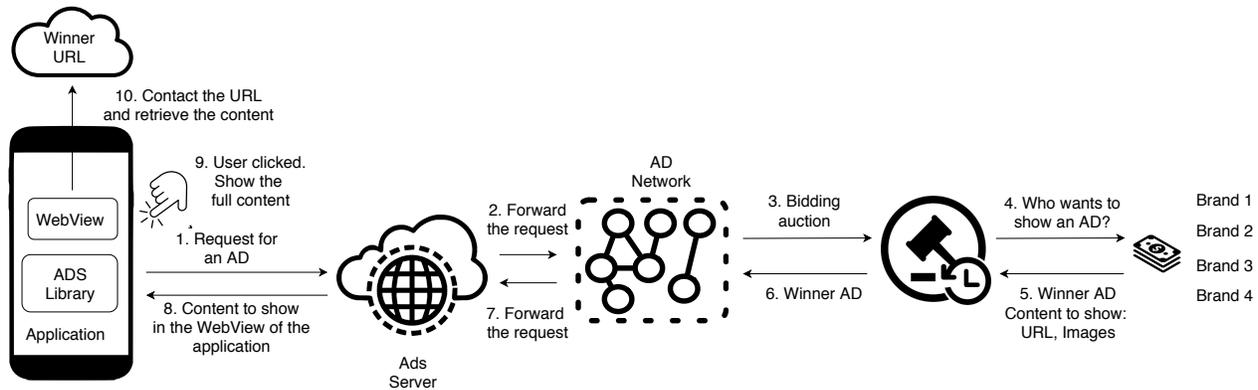
Figure 2: Ad Ecosystem of *individual ad network*

honor the policy. Given the difficulty and missing documentation, we were positively surprised by this high adoption rate. We thus decided to investigate why libraries are enforcing such a difficult part of the policy and not the easier-to-enforce cleartext settings. For these libraries, we performed manual analysis (including source code analysis, when available) to determine how the policy is actually enforced. We found that none of these libraries is implementing SSL/TLS-related operations from scratch nor defining a custom handler for CAs. Instead, they are all relying on core Android framework methods to perform SSL operations, which includes handshake and management of the KeyStores. All these operations are handled by the *Conscrypt* [23] package, which provides Java Secure Socket Extension (JSSE). While this is clearly a positive news, we find it surprising that these popular network libraries do not adhere to arguably more critical cleartext settings.

We also found that *AndroidAsync*, used by thousands of apps, does not support NSP at all. In fact, we found that the mere presence of a `domain-config` node is enough to break the network library, leading to an exception, and thus making it essentially incompatible with the NSP. Table 3, in Appendix, summarizes our findings.

5.4 Disclosure

We disclosed our findings to Google, with an emphasis on the misconfiguration of the SSL Pinning (which may give a false sense of security to inexperienced app developers). We also proposed to extend the AndroidStudio IDE with a linter for the NSP that checks for these misconfigurations and informs the developer about the potential risks. Google acknowledged that this is, in fact, a rather odd configuration. For what concerns the networking libraries not compliant with the actual NSP (see Table 3), we have disclosed our findings to the developers. We are still working towards full bugs fixes.

6 Impact of Advertisement Libraries

Advertisements (ads, in short) play a key role in mobile apps. In this section, we first provide an overview of how advertisement libraries (ad libraries) operate and their complexity, and we then explore the implications for the adoption of the NSP.

Ads are the most important source of income for many app developers, especially when they can be freely downloaded from the Play Store. An app can simultaneously embed one or multiple ad libraries. While the app is running, the ad library retrieves the content of the ads from a remote server and it displays it to the user. Every time an ad is shown to the user, the developer earns a revenue. If the user clicks on the ad, the developer then gets a more substantial revenue. Even though this mechanism is conceptually simple, the actual implementation details and the underlying process are far from trivial. We now quickly discuss the main steps, which are also depicted in Figure 2. First, the developer embeds a given ad library in her application. Then, when the app is running, the ad library contacts its backend server and asks for an ad to be displayed. Depending on the ad library's implementation, this first request can reach one or multiple servers. In case of an *individual ad network*, the library contacts a single server, while in case of an *ads aggregator* the request is sent to multiple servers. The server then forwards the request to its *ad network*, which might be more or less complex. Within the ad network, the *bidding auction* starts. Bidding consists of advertisers (brand) declaring the maximum amount of money they are willing to pay for each impression (or click) of their ad. The winner sends the content of the ad back to the library, and the ad is then displayed in the app, normally within a WebView. Moreover, if the user clicks on the ad, then the *full enriched content* is retrieved from the server of the auction's winner (which is related to the specific ad, and *not* to the ad library itself). The complexity of the ad ecosystem and the interconnection of multiple players—each of which only controls a *portion* of the ecosystem—opens interesting questions related to the NSP. Since the winner of the auction is usually not under

the control of the ad library, the *enriched content* downloaded upon a user's click may be served via HTTP: this aspect makes it interesting to determine how different ad libraries deal with this "uncertainty" on the protocol used by the advertiser. Motivated by these observations, we set out to perform the first systematic analysis of the NSPs defined by ad libraries.

The rest of the section is organized as follows. First, we present the dataset we built for the analysis. Then, we analyze and characterize the NSPs defined by ad libraries, and we show how several of these libraries push app developers to severely weaken their policies, oftentimes justifying these requests with misleading arguments. We then end our discussion with an in-depth case study. We note that, ideally, it would be interesting to perform large-scale and automated analysis over many ad libraries. However, we refrain from performing such study due to ethical concerns: in fact, automatically visiting apps with the mere goal of generating ad impressions that would *not* be seen by real users (or, even worse, automatically clicking on these ads) would generate illegitimate revenues for the app developer (who could be framed as fraudster), and it would damage all the ads ecosystem's parties involved.

6.1 Dataset

To perform this investigation, we built a comprehensive and representative dataset of the most used ad libraries. We choose the Top 29 ad libraries from AppBrain [10] based on the ranking "number of applications." Table 4, in Appendix, summarizes the statistics about the ad libraries.

6.2 Policy Characterization

We investigated whether a given ad library requires a policy modification and of which kind. To identify if a library requires a policy, we start by looking at its official documentation. In case we do not find any reference to the NSP, we then proceed by analyzing the source code of the "reference example app," which is always provided by the ad library developers to show how such a library can be integrated. Among the 29 libraries that we analyzed, we found that *12* of them do require the developer to modify the policy. (The remaining 17 do not require any modification, which suggests that their backend infrastructure is fully compliant with the latest standards and defaults.) One of these is the Facebook ad library, which only requires the developer to specify a configuration for a single domain (see Section 5.2). The other libraries require more invasive modifications, which we discuss next.

Cleartext. Our first finding is concerning: All the 11 libraries require the developer to allow cleartext on her application. *We found that MoPub, HyprMx, HeyZap, Pollfish, AppMediation, and Appodeal do force the developer to completely allow cleartext protocols for all domains*. We also found that *AdColony, VerizonMedia, Smaato, AerServ, and DuApps push the developer to*

adopt the first version of the policy, with similarly negative consequences. These configurations make ineffective any safety net that a NSP may provide. However, we note that these ad libraries may be *required* to ask for this modification since it could be that a given ad framework does not have enough control over the type of URLs (HTTP vs. HTTPS) that are served as part of the ads.

Trusted Anchors. We have identified ad libraries defining a `trust-anchors` node. Even in this case, the findings are concerning: *Appodeal* [20] and *HeyZap* [21] suggest the developer to add *User KeyStore* as trusted, thus providing a venue to perform MITM attacks. Moreover, none of these libraries provide any custom CA, nor ask the developer (or the user) to do so, making this risk completely unnecessary.

Misleading Documentation. We argue that the security repercussions of NSP modifications should be explained and justified to developers so that they can take informed decisions on whether to include a given ad library. However, we found how this "transparency" is not a common practice. After closely inspecting the documentation of the 11 ad libraries mentioned above, we found that *none* of them inform developers of the possible consequences of allowing cleartext protocols or trusting User KeyStores. Some of these libraries simply inform the developers that they *need* to apply their modifications of the NSP in the name of "usability" and to avoid any faulty behavior. Moreover, we identified how *Millenial Media, Smaato, HyprMX, and AerServr* simply ask the developer to copy-paste the provided sample AndroidManifest, without explicitly mentioning the fact that such a sample manifest silently specifies a "usesCleartextTraffic" policy. Even worse, we found how *Du Apps* misleadingly justifies the need to allow cleartext traffic because it is *"required for target SDK28."* We believe that the underlying reason for these problems is that most of these ad libraries found themselves in difficulty due to their infrastructure not being ready to deal with Google's HTTPS everywhere push.

6.3 Ad Libraries in Apps

As previously discussed, we identified some ad libraries that ask developers to weaken their security policy and to allow cleartext. We performed additional experiments that aim at determining how frequently these ad libraries are used within our dataset and whether these apps allow cleartext as part of their NSP.

To detect a third-party library within a given app, we use `LibScout` [12], the state of the art static analysis tool for this kind of task. According to the paper, `LibScout` can detect the inclusion of external libraries within apps even when common bytecode obfuscation techniques are used. `LibScout` supports two types of detection: the first one is based on a simple matching with the package name, while the second one relies on code similarity. By default, it reports only matches that have a similarity of at least 70%. For our experiment, we used the same threshold. Currently, `LibScout` supports only the *Facebook Audience* ad library. We extended it by creating profiles, necessary for the detection, for all the ad libraries that require the developer to

Table 1: The table summarizes the results of the analysis with `LibScout`. *Dataset 1* represents the analysis over 16,324 apps, while *Dataset 2* represents the analysis over the second version of the dataset composed of 108,542 apps.

# Apps with Ad library matched by	Dataset 1	Dataset 2
Package Name (PN)	3,189	9,304
Code Similarity (CS)	2,072	5,918
PN $\wedge \neg$ *CS*	1,158	3,727
CS $\wedge \neg$ *PN*	41	341
PN $\wedge$ *CS*	2,031	5,577
PN $\vee$ *CS*	3,230	9,645

Table 2: The table presents the distribution of the dataset in terms of inclusion of ad libraries (that ask developers to weaken their policy) and whether the apps' NSP allows cleartext.

NSP	Dataset 1		Dataset 2	
	Ads	No Ads	Ads	No Ads
Cleartext	2,891	10,956	7,298	65,455
No Cleartext	339	2,138	2,347	33,442

modify the NSP to allow cleartext. Then, for each of the apps in our datasets, we run `LibScout` for a maximum time of one hour.

We run `LibScout` on the first dataset of 16,324 apps (which specify a NSP), and also on the second "fresher" dataset of 108,542 apps. For the first dataset, `LibScout` was not able to conclude the analysis in time for 8 apps, while it terminated correctly for all the apps in the second dataset. In total, the matching engine was able to identify that 19.7% of the apps belonging to the first dataset (3,230) do have one of the ad libraries that requires cleartext. For the second dataset, instead, it identified 8.8% of apps (9,645) containing at least one of the libraries.

Table 1 summarizes the results. Unfortunately, we suspect that `LibScout` may miss several matches (that is, it does not find libraries even if they are included). In fact, Table 1 shows how the matching results are dominated by the "package name" heuristic, and how only 41 matches for the first dataset, and 341 for the second, were solely due to the similarity analysis engine (i.e., all other matches were already covered by the package name heuristic, hinting that the apps were not obfuscated). We thus remind the reader that, for the numbers reported in this section, the accuracy of these numbers is based on the accuracy of the underlying libraries matching engine, `LibScout`.

We then proceeded by checking how many of the apps identified by `LibScout` effectively have a NSP that allows global cleartext, as defined by the ad libraries. Table 2 summarizes our findings. We note how for the first dataset, 89% of the apps (2,891) embedding an ad library do have a NSP that allows cleartext. However, 11% (339) do not allow it: for these apps, the ads served over HTTP will not be displayed and an Exception is thrown. We also note that even if apps do not use ad libraries, a large portion of them (83%) still use HTTP. Thus, while ad libraries asking developers to weaken their security policy certainly does not help, it does not seem to be the only reason app developers stick to insecure HTTP connections. For the second dataset, we found that, among apps that include an ad library, 75.6% of them (7,298) define a NSP that permits cleartext. The percentage of apps that allow cleartext decreases to 66.1% when considering apps that do not include one of the ad libraries we have checked for.

6.4 Case Study: MoPub

We now present an in-depth analysis of one of the most prominent ad libraries, MoPub [22]. This library is an *individual ad serving platform* used by over 19k applications, some of which have more than 50M unique installations. MoPub is one of those libraries that *requires* an app developer to allow cleartext for her entire app. For this case study, we set out to determine whether this library really had no other choice but to require cleartext on the entire app to properly work. To shed some light, we aimed at monitoring the network requests performed by this ad library at run time. We note that a simple network monitor on the traffic generated by the entire app is not enough: by just observing network traces, it would be very challenging to determine which traffic has been generated by the ad library and which by unrelated components of the app.

Thus, we developed an instrumentation framework that records all network activities and, moreover, hooks the network `Socket.connect` API (by using Frida [38]). This API is the lowest-level API used for any HTTP or HTTPS connection and it provides the target domain name and the port. Every time the API is invoked, we perform a stack trace inspection to determine which package has originated the call: this setup allows us to match which component (i.e., library) of the app initiated the network request.

Due to the ethical concerns mentioned earlier, we limited ourselves to a very small-scale experiment: we opted to select and analyze only one representative app, *Hunter Assassin* [44], an action game with more than 50M installations. This app embeds MoPub and specifies a NSP that reflects MoPub's documentation. For the experiments, we executed the app 10 times, with each execution lasting 10 minutes. Due to ethical concerns, we opted to not use automatic UI stimulation techniques, but we performed this analysis step manually, by just simulating the interaction of a "real" user. This approach allows us to avoid generating excessive traffic and damage the app developer's reputation and ad libraries.

During the analysis, our instrumentation framework detected that the MoPub library initiated connections to 83 unique domains. (For this experiment, we discarded the domain names reached by other components of the app.) Surprisingly, for 82 domains (out of 83) the connection was actually established using HTTPS, the only exception loaded over HTTP being

an image, retrieved from a MoPub server. Even though this HTTP connection would be blocked by a non-permissive cleartext policy, we do not believe this is *the* core reason why MoPub requires the policy to allow cleartext for the entire app. According to the MoPub documentation, it requires HTTP because it *may* need to serve ads via HTTP—and to do so, it asks the app developer to weaken the policy for the entire app.

We believe this to be a clear violation of the principle of least privilege, as the ad library should allow cleartext for its own connections, without interfering with the rest of the app. However, we note that this current situation is not solely fault of the ad library: with the current policy format, it would be impossible to enumerate all possible domain names that the ad library should be able to reach since this list is not known in advance (and since the NSP cannot be changed at run-time). We identified a conceptual limitation: the current policy format allows developers to specify policies *per domain*, but we believe a better abstraction for policy specification to be *per package*. In an ideal world, the ad library should be able to express that *only the connections that are initiated by the MoPub library itself* should be subject to use cleartext, without the need of weakening the rest of the app. Guided by these insights, we designed and implemented a drop-in extension to the NSP that would address this concern. We discuss this proposal in the next section.

7 Network Security Policy Extension

As previously discussed, third-party libraries can significantly weaken the NSP of an app, and ad libraries actually often do so. In some scenarios, however, it is very challenging for ad libraries to "do better." In fact, the complexity of the ad ecosystem may make it impossible, for example, to know in advance which domain names require HTTP connections, thus leaving the ad library developers to ask to allow cleartext for the entire app. We believe the current format of the policy is fundamentally limited. The current policy allows developers to specify different policies at the granularity level of domain names: we argue that, in some scenarios (e.g., ad libraries), this is the wrong abstraction level.

This section discusses our proposal for an extension of the NSP format to allow for the specification of policies at a different granularity: app components, identified by their package names.

Our New Extension. The core idea behind the extension is to allow a developer to bind a specific policy to a specific package name(s). To this end, we introduce a new XML node, `package-config`, which allows developers to specify custom policies for specific external libraries, without the need to modify (and negatively affect) the policy of the main app. To ease the explanation, consider the following concrete example:

```
<base-config cleartextTrafficPermitted="false" />
<package-config><!--introduced by our extension-->
  <package name="com.adlib.unsafe"
    cleartextTrafficPermitted="true"/>
</package-config>
```

This policy specifies that, by default, all HTTP traffic should be blocked. However, it would allow HTTP connections if they are initiated by the `com.adlib.unsafe` ad library. Note how the ad library can now support occasional HTTP connections even without knowing the list of domain names a priori and, more importantly, without affecting the policy of the app.

Implementation. We implemented this new extension by modifying the `isCleartextTrafficPermitted` API to make it aware of the XML policy node. Our modification performs stack trace inspection to determine which package name has initiated the call. For each package name appearing in the stack trace, we then check whether the NSP contains a custom policy for a specific package name: if yes, we use that policy. Otherwise, we apply the default. In case the connection should *not* be allowed, our implementation raises a `RuntimeError`, indicating a policy violation.

Adoption & Backward Compatibility. Our extension can be trivially adopted by app developers and network libraries. In fact, since we modify an API that all these libraries already invoke—and that was a key design choice—they can enjoy the benefits of our policy without the need to make any modification. We also note that our extension is fully backward compatible and it can act as a drop-in replacement of the old version. In fact, apps and policies that are not "aware" about our extension are supported exactly the same as before.

Performance Considerations. We implemented our extension on a Pixel 3A running Android Pie (pie-qpr3-b-release). Our patch consists of less than 30 lines of code and modifies only two components of the Android framework (the policy parser and the `isCleartextTrafficPermitted` API). We measured the overhead of our extension with a microbenchmark: we wrote an app that performs 1,000 HTTP requests using the *OkHttp3* library. We then run the app 100 times, with and without our modifications, and we compute the difference. The average execution time of the `isCleartextTrafficPermitted` API, without our modification, is *0.004 ms* with a standard deviation of *0.006 ms*. The average execution time of the same API with our modification is instead *0.30 ms*, with a standard deviation of *0.094 ms*. We believe that the overhead of our defense mechanism is negligible, especially when compared to the overhead incurred by network I/O operations.

Limitations. Even though our implementation raises the security bar of the current Network Security Policy, we acknowledge that it currently suffers from some limitations. First, it is important to mention that, since we operate with the same threat model of the actual NSP, we do not protect the application against malicious third-party libraries that want to evade the policy defined by the developer. We note that this affects the standard NSP as well: in fact, a malicious library can bypass even the strictest security policy by performing network connections with its "custom" API or by using native code.

A second limitation relates to the fact that we rely on the stack trace to identify which component initiated the

network connection. We acknowledge that there may be benign situations where the stack trace cannot be fully trusted and there might be the risk of losing the real "caller," for example, when using dynamic code loading or threading with worker threads. A very detailed analysis of the potential problems of using the stack trace to perform "library compartmentalization" has been studied in FlexDroid [40]. Even if the current threat model of FlexDroid is considering malicious libraries, we believe that their proposal of a secure inter-process stack trace inspection combined to our defense mechanism might create a full-fledged implementation to tackle the compartmentalization problem.

To conclude, we currently support only the `cleartextTrafficPermitted` attribute. However, note that some features already provide a sufficient granularity and do not need to be sandboxed on a "per-package" basis. For example, the certificate pinning feature already creates a sort of "per-site sandbox."

8 Related Work

There are several areas of works that are relevant to this paper: Network Security, the dangerousness of "code reuse," and advertisements.

Network Security. A concept similar to the NSP has been first introduced by Fahl et al. [26]: this work proposed a completely new approach to handle SSL security, allowing developers to easily define different SSL configurations and options, like certificate pinning, just by using a XML policy. Thus, [26] completely prevents the developer to write any code responsible of handling the validation and verification of a given certificate, addressing multiple problems at their roots.

Another group of works focuses on the risks of using unencrypted connections. Vanrykel et al. [46] study how apps send unique identifiers over unencrypted connections exposing the user to privacy threats, while [16, 37] show how several apps are vulnerable to remote code injection due to code updating procedures over HTTP.

Several works evaluate the adoption of secure connections among apps: Razaghpanah et al. [39] measured the adoption of different libraries performing SSL/TLS operations by fingerprinting their handshake. Oltrogge et al. [34], instead, measured the adoption of certificate pinning and, by surveying the developers they discovered that the implementation of pinning is considered complex and hard to correctly implement.

Other works focus on identifying SSL problems among apps. One such example is by Fahl et al. [25], which applied static code analysis and found multiple applications with SSL/TLS code that is potentially vulnerable to MITM attacks. Hubbard et al. [30] and Onwuzurike et al. [35], instead, applied a combination of static and dynamic analysis to identify SSL vulnerabilities in popular Android apps.

To conclude, Damjan et al. [15] propose a new defense mechanism to overcome the problem of broken SSL/TLS implementations named *dynamic certificate pinning*, while Zhao et al. [51]

discuss several possible counter-measures against SSLStrip.

Code Reuse. Several works highlighted how developers rely on online platforms like StackOverflow for their development process. Linares-Vásquez et al. [47] analyzed more than 213k questions on StackOverflow (related to Android) and built a system to pair a given snippet of code of StackOverflow with a given snippet of code within the Android framework. Their work showed how developers ask questions and change their code once the behavior of a given API changes. Fischer et al. [27], instead, measured the proliferation of security-related code snippets from StackOverflow in Android apps available on Google Play. [27] showed how more than 200k apps contain copy-pasted security-related code snippets from StackOverflow. A similar work, not focused on Android, is from Verdi et al. [48] in which they investigated security vulnerabilities in C++ code snippets shared on StackOverflow. They showed how 2,859 GitHub projects are still affected by vulnerabilities introduced by vulnerable C++ code snippet copied from StackOverflow.

Advertisements. Ads on Android have been evaluated both in terms of privacy and security. The first category of works studies ad libraries to identify the privacy implications for the user. Book et al. [13] tracked the increase in the use of ad libraries among apps and highlight how the permissions used by these libraries may pose particular risks to user privacy. Son et al. [42] demonstrate how malicious ads can leak the PII of the user. Stevens et al. [43] instead show how users can be tracked across ad providers due to the amount of personal information sent from the ads libraries and expose how these libraries checked for permissions beyond the required ones to obtain more PII.

The second group of works, instead, focuses mostly on the security impact of ad libraries and proposes different solutions to achieve privilege separation for applications and ads. AdDroid [36] proposes a new advertisement API to separate privileged advertising functionality from the app. AFrame [50] and AdSplit [41], instead, propose a different approach to let ad libraries run in a process separate from that of the application.

9 Conclusion

In this work, we performed the first large-scale analysis of Network Security Policies on the Android ecosystem and we systematically explored the adoption of this new defense mechanism by Android apps. Our analysis shows how developers are still allowing full cleartext on their application. We investigated why insecure communications are still vastly used by applications and we determined one of the root causes to be related to the complex ad ecosystem. Guided by these findings, we designed and implemented a drop-in extension on the actual NSP, which allows developers to specify a "per-package" policy, so that they can embed third-party ad libraries without needing to compromise their app's security. We hope this work provides useful insights to speed up Google's "HTTPS Everywhere on Android" effort.

Acknowledgements

References

[1] RFC 5280. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL). `http://tools.ietf.org/html/rfc5280`. 2008, Accessed: June, 2020.

[2] Heather Adkins. An update on attempted man-in-the-middle attacks. `https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html`. 2011, Accessed: June, 2020.

[3] Klyubin Alex. An Update on Android TLS Adoption. `https://security.googleblog.com/2016/04/protecting-against-unintentional.html`. 2016, Accessed: June, 2020.

[4] HCL Technologies Alon Galili of Aleph Research. Cordova-Android MiTM Remote Code Execution, CVE-2017-3160. `https://alephsecurity.com/vulns/aleph-2017013`. 2017, Accessed: June, 2020.

[5] Official Documentation Android Developers. Android Manifest application. `https://developer.android.com/guide/topics/manifest/application-element.html?#usesCleartextTraffic`. 2019, Accessed: June, 2020.

[6] Official Documentation Android Developers. Network security configuration. `https://developer.android.com/training/articles/security-config`. 2019, Accessed: June, 2020.

[7] Official Documentation Android Developers. NetworkSecurityPolicy isCleartextTrafficPermitted, API. `https://developer.android.com/reference/android/security/NetworkSecurityPolicy.html#isCleartextTrafficPermitted()`. 2016, Accessed: June, 2020.

[8] Platform Documentation Android Developers. Android 8.0 Behavior Changes. `https://developer.android.com/about/versions/oreo/android-8.0-changes`. 2020, Accessed: June, 2020.

[9] AndroidRank. AndroidRank, open android market data since 2011. `https://www.androidrank.org`. 2011, Accessed: June, 2020.

[10] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Ad Networks. `https://www.appbrain.com/stats/libraries/ad-networks`. 2011, Accessed: June, 2020.

[11] AppBrain. AppBrain: Monetize, advertise and analyze Android apps. Network Libraries. `https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries`. 2011, Accessed: June, 2020.

[12] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[13] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. 2013.

[14] Chad Brubaker. Changes to Trusted Certificate Authorities in Android Nougat. `https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html`. 2016, Accessed: June, 2020.

[15] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. Pin it! Improving Android network security at runtime. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, 2016.

[16] Hyunwoo Choi and Yongdae Kim. Large-Scale Analysis of Remote Code Injection Attacks in Android Apps. 2018.

[17] Catalin Cimpanu. Over 58,000 Android users had stalkerware installed on their phones last year. `https://www.zdnet.com/article/over-58000-android-users-had\-stalkerware-installed-on-their\-phones-last-year/`. 2019, Accessed: June, 2020.

[18] MDM contributors. Web technology for developers: Strict-Transport-Security. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security`. 2020, Accessed: June, 2020.

[19] DataTheorem. TrustKit Android: Easy SSL pinning validation and reporting for Android. `https://github.com/datatheorem/TrustKit-Android`. 2016, Accessed: June, 2020.

[20] Appodeal Android SDK Developer. Appodeal Android SDK. Android SDK Integration Guide. `https://wiki.appodeal.com/en/android/2-6-4-android-sdk-integration-guide`. 2019, Accessed: June, 2020.

[21] HeyZap Android SDK Developer. HeyZap Android SDK. `http://web.archive.org/web/20190615131844/https://developers.heyzap.com/docs/android_sdk_setup_and_requirements`. 2019, Accessed: June, 2020.

[22] MoPub SDK Developer. Integrate the MoPub SDK for Android. `https://developers.mopub.com/publishers/android/get-started/`. 2020, Accessed: June, 2020.

[23] Android Developers. AOSP Design Architecture: Conscrypt. `https://source.android.com/devices/architecture/modular-system/conscrypt`. 2020, Accessed: June, 2020.

[24] Android Developers. Official Documentation NetworkSecurityPolicy, API. `https://developer.android.com/reference/android/security/NetworkSecurityPolicy`. 2016, Accessed: June, 2020.

[25] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In *CCS '12*, 2012.

[26] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *CCS '13*, 2013.

[27] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Gülsüm Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of CopyPaste on Android Application Security. 2017.

[28] Hogben Giles and Idika Nwokedi. Protecting against unintentional regressions to cleartext traffic in your Android apps. `https://android-developers.googleblog.com/2019/12/an-update-on-android-tls-adoption.html`. 2019, Accessed: June, 2020.

[29] Leonid Grustniy. What's wrong with "legal" commercial spyware. `https://www.kaspersky.com/blog/stalkerware-spouseware/26292/`. 2019, Accessed: June, 2020.

[30] John Hubbard, Ken Weimer, and Yu Li Chen. A study of SSL Proxy attacks on Android and iOS mobile applications. 2014.

[31] John Kozyrakis. CVE-2016-2402. `https://koz.io/pinning-cve-2016-2402/`. 2016, Accessed: June, 2020.

[32] MWR F-Secure Lab. Paypal Remote Code Execution, CVE-2013-7201, CVE-2013-7202. `https://labs.f-secure.com/advisories/paypal-remote-code-execution/`. 2013, Accessed: June, 2020.

[33] John Leyden. Inside 'Operation Black Tulip': DigiNotar hack analysed. `https://www.theregister.co.uk/2011/09/06/diginotar_audit_damning_fail/`. 2011, Accessed: June, 2020.

[34] Marten Oltrogge, Yasemin Gülsüm Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To Pin or Not to Pin-Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security Symposium*, 2015.

[35] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *WISEC*, 2015.

[36] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David A. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. 2012.

[37] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Krügel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS*, 2014.

[38] Ole André Vadla Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. `https://frida.re/docs/android/`. 2020, Accessed: June, 2020.

[39] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Philippa Gill. Studying TLS Usage in Android Apps. In *Proceedings of the Applied Networking Research Workshop*, 2018.

[40] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS*, 2016.

[41] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *USENIX Security Symposium*, 2012.

[42] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What Mobile Ads Know About Mobile Users. In *NDSS*, 2016.

[43] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Lee Erickson, and Hao Chen. Investigating User Privacy in Android Ad Libraries. 2012.

[44] Ruby Game Studio. Hunter Assassin. `https://play.google.com/store/apps/details?id=com.rubygames.assassin`. 2020, Accessed: June, 2020.

[45] Android Security Team. Google Report: Android Security 2014 Year in Review. `https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf`. 2014, Accessed: June, 2020.

[46] Eline Vanrykel, Gunes Acar, Michael Herrmann, and Claudia Díaz. Exploiting Unencrypted Mobile Application Traffic for Surveillance Technical Report. 2017.

[47] Mario Linares Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger stack overflow discussions? a study on the Android SDK. In *ICPC 2014*, 2014.

[48] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Motlagh. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples, 2019.

[49] Ryan Welton. Remote Code Execution as System User on Samsung Phones. `https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user\-on-samsung-phones`. 2015, Accessed: June, 2020.

[50] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: isolating advertisements from mobile applications in Android. In *ACSAC '13*, 2013.

[51] Yan Zhao, Youxun Lei, Tan Yang, and Yidong Cui. A new strategy to defense against SSLStrip for Android. In *2013 15th IEEE International Conference on Communication Technology*, 2013.

Appendix

Network Libraries Compliance

Table 3: The table summarizes the results of the analysis on network libraries. For each library, it first presents statistics about the number of applications using the given library and how many downloads has the top downloaded application. Then, for each of the tested feature, we mark with a ✓ when the library passes all the testcases, ✗ otherwise. The last column represents whether an application is honoring the entire policy (●), only a subset of features (◐) or none of them (○). For URLConnection, statistics are not available on AppBrain.

Networking Library	# of Apps	Top App Downloads	Cleartext	Certificate Pinning	Trust Anchors	Compliant
Retrofit	> 104k	1 B	✓	✓	✓	●
Volley	> 66k	5 B	✓	✓	✓	●
OkHttp	> 39k	5 B	✓	✓	✓	●
AndroidAsyncHTTP	> 22k	100 M	✗	✓	✓	◐
AndroidAsync	> 7k	100 M	✗	✗	✗	○
FastAndroidNetworking	> 3k	100 M	✓	✓	✓	●
HttpClientAndroid	∼ 1,000	100 M	✗	✓	✓	◐
BasicHttpClient	∼ 1,000	100 M	✓	✓	✓	●
Robospice	∼ 1,000	10 M	✓	✓	✓	●
URLConnection	N/A	N/A	✓	✓	✓	●

A Complete Network Security Policy

```
<network-security-config>
    <domain-config
      cleartextTrafficPermitted="false">
      <domain includeSubdomains="false">
      android.com</domain>
      <pin-set expiration="2020-12-12">
        <pin digest="SHA-256">YZPgTZ+woNCCCIW3LH2CxQeLzB/1m42QcCTBSdgayjs=
        </pin>
      </pin-set>
    </domain-config>
    <debug-overrides>
      <trust-anchors>
        <certificates src="system"/>
        <certificates src="@raw/custom_cert"/>
      </trust-anchors>
    </debug-overrides>
</network-security-config>
```

Figure 3: The policy lacks a `base-config`. Thus, its configuration changes according to the API level. For example, if the app targets the API level 28, the policy will deny all cleartext protocols and use only the system CAs. The policy also defines a different security mechanism for the *android.com* domain (but not for its subdomains). In particular, the policy specifies that the application should reach the domain only via HTTPS and only with a specific certificate (i.e., it implements certificate pinning). The policy also defines an expiration date for this certificate. Moreover, when the application is compiled in *debug mode*, network connections can be trusted if they are signed with CAs defined within the system KeyStore or with a custom, hardcoded CA *"custom_cert"*. Also, no certificate pinning is enforced.

Example of Real Network Security Policy

```
<domain-config>
  <domain>example.com</domain>
    <pin-set>
      <pin digest="SHA-256">HASH</pin>
    </pin-set>
</domain-config>
```

(a)

```
<domain-config>
  <domain>valid_domain</domain>
    <pin-set>
      <pin digest="SHA-256">BBBBB..BBBBBB</pin>
    </pin-set>
</domain-config>
```

(b)

Figure 4: These policies represent two real cases found on our dataset. On the policy (a), it is possible to see how the developer enforced the certificate pinning on the *example.com* domain, while in the policy (b) the developer enforced certificate pinning with a wrong certificate formed of only "B."

Advertisement Libraries

Table 4: The table summarizes the results of the analysis on advertisement libraries. For each library, it presents statistics about the number of applications using the given library, how many downloads has the top downloaded application, and if it requires a modification of the NSP. For AppMediation, the statistics are not available anymore: however, the required policy can be found at `https://github.com/appmediation/Documentation/wiki/Android-Project-Setup`

Ad Library	# of Apps	Top Apps Downloads	Requires NSP Modification
AdMob	> 464k	1B	
Facebook Audience Network	> 96k	500M	✓
Unity	> 67k	50M	
AppLovin	> 34k	100M	
Chartboost	> 30k	1B	
Startapp	> 29k	100M	
AppsFlyer	> 29k	500M	
AdColony	> 24k	100M	✓
Vungle	> 20k	100M	
MoPub	> 19k	1B	✓
Ironsource	> 19k	50M	
Amazon Mobile Ads	> 13k	500M	
Tapjoy	> 11k	100M	
InMobi	> 11k	100M	
Pollfish	> 9k	10M	✓
AppNext	> 8k	100M	
Adjust	> 8k	1B	
HeyZap	> 7k	100M	✓
Smaato	> 4k	100M	✓
Fyber	> 4k	100M	
Millennial Media	> 3k	500M	✓
MyTarget	> 3k	100M	
Appodeal	> 3k	50M	✓
Kochava	> 2k	100M	
AerServ	> 2k	100M	✓
Tenjin	~ 1,000	100M	
HyprMX	~ 1,000	100M	✓
DU Ad	~ 500	100M	✓
AppMediation	N/A	N/A	✓