Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications

Marten Oltrogge* *marten.oltrogge@cispa.saarland*
Nicolas Huaman† *huaman@sec.uni-hannover.de*
Sabrina Amft† *amft@sec.uni-hannover.de*
Yasemin Acar† *acar@sec.uni-hannover.de*
Michael Backes* *backes@cispa.saarland*
Sascha Fahl† *fahl@sec.uni-hannover.de*

*CISPA Helmholtz Center for Information Security
†Leibniz University Hannover

Abstract

Android applications have a long history of being vulnerable to man-in-the-middle attacks due to insecure custom TLS certificate validation implementations. To resolve this, Google deployed the Network Security Configuration (NSC), a configuration-based approach to increase custom certificate validation logic security, and implemented safeguards in Google Play to block insecure applications.

In this paper, we perform a large-scale in-depth investigation of the effectiveness of these countermeasures: First, we investigate the security of 99,212 NSC settings files in 1,335,322 Google Play apps using static code and manual analysis techniques. We find that 88.87% of the apps using custom NSC settings downgrade security compared to the default settings, and only 0.67% implement certificate pinning. Second, we penetrate Google Play's protection mechanisms by trying to publish apps that are vulnerable to man-in-the-middle attacks. In contrast to official announcements by Google, we found that Play does not effectively block vulnerable apps. Finally, we performed a static code analysis study of 15,000 apps and find that 5,511 recently published apps still contain vulnerable certificate validation code.

Overall, we attribute most of the problems we find to insufficient support for developers, missing clarification of security risks in official documentation, and inadequate security checks for vulnerable applications in Google Play.

1 Introduction

Studying the security of Android applications has a long history [35] and was heavily influenced by the seminal paper by Enck et al. in 2011 [52]. A myriad of investigations demonstrated that developers struggle with different aspects of implementing Android application security mechanisms correctly [46, 50–52, 73, 78]. The number of affected users of Android applications vulnerable to different types of attacks due to incorrect security implementations goes into billions [31].

While developers fight with many different security challenges, custom TLS certificate validation security received attention early on in 2012 [54, 56] and has become a hotly debated topic over the years [47, 48, 56, 58, 74, 76, 77, 84, 85]. The problem not only affects Android applications but turns out to be a broader issue in secure programming [38, 42, 69]. Researchers proposed different countermeasures which all focus on simplifying the process of implementing non-standard TLS certificate validation such as certificate or public key pinning or the secure use of self-signed certificates for applications under development [56, 74, 85].

However, the problem not only received attention from academia. Google introduced countermeasures and novel mechanisms for developers in Android and added further security policies and safeguards to Google Play (cf. Table 1). Their goal was to establish new and safer defaults such as enforcing TLS for all network connections by default and blocking vulnerable apps and updates from Google Play.

Therefore, Google introduced a significant change in Android 7 in 2016: The Network Security Configuration (NSC) [19] allows developers to implement custom certificate validation logic using an XML configuration file, instead of requiring custom code.

Additionally, Google Play announced novel security policies and safeguards in 2016 and 2017 [66–68]. They prohibit new apps and updates to include insecure certificate validation logic. While previous work (e.g. [70, 75, 80, 86] found vulnerable apps in Google Play that were published after 2016, our study is the first detailed analysis of Google Play's safeguard efficacy.

Although the goal of all introduced changes is to improve TLS security for Android applications and fix the disastrous circumstances that researchers uncovered in 2012 [54] and 2013 [56], the efficacy and success of this undertaking has not yet been investigated in-depth. However, incidents illustrate that Network Security Configuration is not a guarantee for secure certificate validation logic in Android apps: In 2019, Google's official Gmail app for Android had come with an insecure NSC setting that opened the possibility for a MitMA

via user-installed CAs. This vulnerability affected 43% of the Android ecosystem [39].

The overall goal of this work is to investigate the current status of TLS certificate validation security in Android apps.

To the best of our knowledge, we provide the first large-scale and in-depth evaluation of the success of Android's NSC approach combined with an analysis of the new security policies and safeguards in Google Play. We also revisit the security of custom certificate validation implementations in Android apps as performed by Fahl et al. [54]. Overall, we make the following contributions:

NSC adoption and security. We measure the adoption of the NSC in 1,335,322 free current Android apps from Google Play, and find that 99,212 apps include custom NSC settings. For these apps, we evaluate the security of their custom NSC settings and find that more than 88.87% of them weaken security by downgrading safe-defaults. In contrast, only 0.67% implement certificate pinning. Our findings illustrate that certificate validation remains a challenging task for developers and requires further attention from the security research community and industry. We report and discuss this contribution in Sections 4 and 4.1.

Efficacy of Google Play Safeguards. We perform multiple experiments to evaluate the efficacy of Google Play TLS security policies and safeguards. We find that Google Play only catches trivial insecure certificate validation code but still accepts most of the dangerous code already found in previous work in 2012 [54, 58]. We replicate work by Fahl et al. [54] and find that out of 15,000 current Android apps in Google Play more than 5,511 contain custom certificate validation code that is vulnerable to MitMAs. These findings are in stark contrast with Google's official statements [66–68] and demonstrate the importance of further research in this area. We report and discuss this contribution in Section 5.

Discussion and Recommendations. Based on our findings, we provide an in-depth discussion of the successes and failures of the NSC approach and Google Play's security policies and safeguards. We illustrate recommendations to improve TLS certificate validation security in future Android versions.

2 Background on TLS and Android

TLS is the most widely deployed network protocol to secure communication channels between clients and servers [36, 82, 83]. It provides confidentiality, integrity, and authenticity for information shared between network end-points and can prevent active and passive MitMAs. While mutual authentication for clients and servers is supported, in most cases only the server's identity is verified. A server is considered trustworthy if the certificate was issued by a trusted certificate authority (CA) for the correct hostname and is still valid [1]. Most modern operating systems include a pre-installed list of trusted root CA certificates. As of June 2020 on Android this list contains 138 entries [4]. While Android correctly validates TLS certificates signed by one of those 138 CAs by default, developers may choose to create their validation logic for several reasons, such as using a custom CA [54]. Before the introduction of NSC, developers had to implement custom certificate validation logic using Android APIs [14, 30, 32]. However, using custom code commonly leads to vulnerabilities [54, 58], such as failing to correctly implement practices like certificate pinning or leaving custom code intended for debugging in production code. Even when putting considerable effort into secure certificate validation implementations, the Android TLS API makes it unnecessarily complicated for developers to implement secure certificate validation (cf. [54]). For example, before Android 4.2, there was no proper API that returned the trusted certificate chain as constructed by the system's certificate validation routines. Hence, attackers were able to manipulate the certificate list as presented by the server. This shortcoming made the implementation of correct CA certificate pinning particularly difficult and made many pinning implementations in the wild vulnerable to MitMAs [71], affecting both app developers as well as libraries such as OkHttp's CertificatePinner [8] [34].

To reduce the threats accompanying insecure implementations, Google introduced significant changes for X.509 certificate validation. We categorize changes into the introduction and updates of NSC and security policy changes and safeguards in Google Play. Table 1 illustrates important changes in chronological order.

2.1 Network Security Configuration

With the release of Android 7, Google introduced the Network Security Configuration (NSC) [19]. NSC supports certificate pinning, custom CA certificates and debugging flags, both globally for all network connections or for specific domains [19].

Figure 1 gives an overview of the structure of an NSC file and how the different features can be combined in `<base-config>`, `<domain-config>` and `<debug-overrides>` sections. Below we provide details for the NSC details that are relevant for our work.

Cleartext Traffic Support This flag can be used to enforce HTTPS or allow HTTP for network connections. Developers can make global or domain specific configurations. Starting with Android 9, cleartext traffic via HTTP is not permitted by default anymore [45]. Instead, HTTPS is used by default [63]. Developers can set the `cleartextTrafficPermitted` flag if they want to enable HTTP (cf. Listing 3 in the Appendix) [21]. Alternatively, developers can configure cleartext traffic support in the application manifest by setting the `android:usesCleartextTraffic` attribute [6]. Since An-

[1] The entire X.509 certificate validation process is much more complex, but left out here for brevity. We refer the interested reader to [43].

Table 1: Chronological overview of TLS-related events in the history of Android:

		Date	Android Version	Description
🤖	1	2015-10-05	Android 6 (API 23)	Android introduces the "android:usesCleartextTraffic" flag for Manifest files, and removes the Apache HTTP Client library [33, 59, 64].
▶	2	2016-05-17		Google Play blocks apps containing unsafe implementations of the X509TrustManager interface [67].
🤖	3	2016-08-22	Android 7 (API 24)	Android introduces NSC, distrusts user-installed certificates, and ignores the "android:usesCleartextTraffic" flag in case a NSC file is available [44, 60].
▶	4	2016-11-25		Google Play blocks apps containing unsafe implementations of the onReceivedSslError method in WebViews [66].
▶	5	2017-03-01		Google Play blocks apps containing unsafe implementations of the HostnameVerifier interface [68].
🤖	6	2017-08-21	Android 8 (API 26)	Android adds support for the "cleartextTrafficPermitted" flag for the WebView class [61].
▶	7	2018-08-01		New apps need to target at least Android 8; makes new safe defaults introduced with Android 7 (2016-08-22) and Android 8 (2017-08-21) [49, 65] available to those apps.
🤖	8	2018-08-08	Android 9 (API 28)	Sets "cleartextTrafficPermitted" to false; enforces HTTPS connections by default. Developers can revert this for specific domains or globally in NSC) settings [63].
▶	9	2018-11-01		App updates need to target at least Android 8; makes new safe defaults introduced with Android 7 (2016-08-22) and Android 8 (2017-08-21) [49, 65] available to existing apps.
▶	10	2019-08-01		New apps need to target at least Android 9; makes new safe defaults introduced with Android 9 (2018-08-08) [49, 65] available to those apps.
▶	11	2019-11-01		Updates need to target at least Android 9; existing apps benefit from new safe defaults introduced with Android 9 (2018-08-08) [49, 65].

🤖 Affects Android OS & NSC – ▶ Affects Google Play security policy & safeguards.

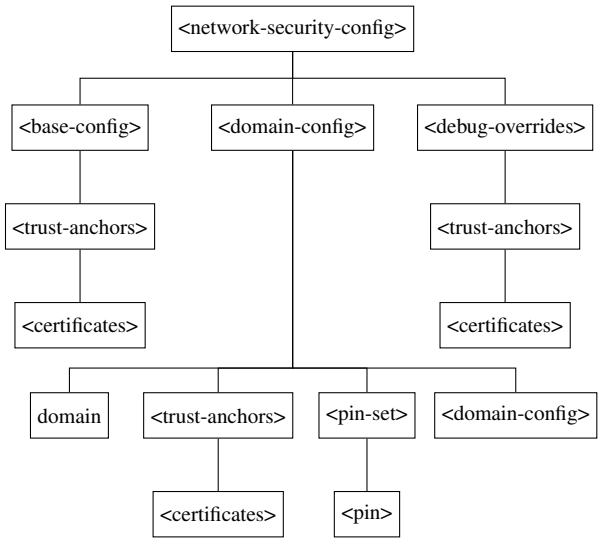

Figure 1: NSC files contain `<base-config>`, `<domain-config>` and `<debug-overrides>` configurations, including custom CA (`<trust-anchors>`) and certificate pinning (`<pin-sets>`) configurations. Cleartext traffic can be permitted or forbidden using the `clearTextTrafficPermitted` flag globally for specific domains.

droid 9, the value is *true* by default. However, it is only honored if no NSC file is provided by the developer.

Certificate Pinning Allows developers to implement certificate pinning [53]. Connections can then only be established if at least one certificate from the server's certificate chain matches any of the registered pins. In contrast to before Android 7, developers do not need to write custom Android code. Developers need to specify expected pinning information inside `<pin>` tags within the `<pin-set>` environment.

Custom Trust Anchors Allows developers to customize the set of trusted CA certificates – e.g., distrusting pre-installed system CA certificates, introducing additional CA certificates, or allowing user-installed CA certificates – for production purposes. As of Android 7, user-installed CA certificates are no longer trusted roots by default. Trust is instead limited to the set of pre-installed system root CA certificates [44, 60]. However, developers can re-enable user-installed certificates by setting the `user` flag (cf. Listing 4), which is a security downgrade comparable to the situation before Android 7.

Debug Settings Allow developers to configure CA certificates – e.g., locally issued or self-signed certificates – for debugging purposes. In contrast to manually implemented code to switch between debug and production logic, it is not possible to have debug settings active in production when publishing apps in Google Play. [28]

Limits of NSC The introduction of NSC did not come along with the deprecation, suspension, or even removal of certificate validation APIs in the Android SDK. Developers are still allowed to write the same erroneous certificate validation code as in earlier Android versions. This is particularly critical since custom certificate validation code overrides NSC settings in some cases (e.g. a vulnerable `TrustManager` implementation makes NSC certificate pinning configurations useless).

2.2 Google Play

In addition to NSC for Android, Google Play implemented a set of policy changes and safeguards.

In 2016 and 2017, Google added safeguards that prevented new apps or app updates to include unsafe `X509Trustmanager` and `HostnameVerifier` interfaces and `onReceivedSslError` methods in WebViews. Google did not provide further details of the safeguards. However, they are executed as part of an app's review process before publishing the app [62]. Since August 2018, Google Play has only accepted apps and updates that target Android 8 [49], which enforces that user-installed certificates are not trusted by default. From late 2019 new apps and updates have been forced to target Android 9 or higher and therefore enforced HTTPS by default [65].

3 Related Work

In this section, we discuss related work regarding measurement studies of insecure TLS certification validation code in Android apps.

In 2012 Fahl et al. [54] analyzed 13,500 popular, free Android apps and found 8% to be susceptible to Man-in-the-Middle-attack (MitMA)s because of insecure TLS certificate validation code. In follow-up work in 2013, Fahl et al. [56] extended their previous analysis to iOS and manually investigated 1,009 applications. They reported that 14% of the apps suffer from similar issues as apps on Android.

Like Fahl et al., Georgiev et al. [58] uncovered a variety of vulnerabilities in TLS certificate verification logic in non-browser software, including mobile apps in 2012. As root causes, they identified poorly designed APIs which confused developers, as well as a lack of safe defaults. In 2014 Sounthiraraj et al. [84] presented SMV-HUNTER, an automated, large-scale analysis tool utilizing a combination of static and dynamic analysis to detect vulnerabilities in the certificate validation logic of Android applications. They performed a study of 23,418 apps, identified 1,453 as potentially vulnerable, and were able to confirm this for 726. In 2015, Onwuzurike and De Cristofaro [77] conducted static and dynamic analyses on 100 popular Android apps and found 32 to implement unsafe TLS certificate validation logic. Furthermore, 91 applications were vulnerable if attackers installed root CAs on a victim's device. In 2015 He et al. [69] presented SSLINT, a tool to detect incorrect use of TLS APIs. They found 27 previously unknown TLS-related vulnerabilities in Ubuntu applications. Fischer et al. [57] classified security-related code snippets from the platform Stack Overflow and assessed their prevalence in Android applications in 2017. They found the most dominant insecure code to be related to unsafe custom TLS. While they could not determine whether or not developers directly copied detected code snippets from Stack Overflow, the authors argue that the platform has a significant impact and responsibility due to its popularity. Razaghpanah et al. [81] conducted a dynamic network traffic analysis with data for 1,364,420 TLS handshakes from 7,258 Android apps using the the Lumen Privacy Monitor framework for 5,000 users in 2017. They find that 2% of the apps in their data set implement custom certificate validation logic.

In contrast to the previous work above, our work focuses on the security of custom NSC settings in deployed Android apps.

Oltrogge et al. [75] analyzed 13 online application generators for Android, of which six failed to implement TLS certificate validation code correctly in 2018. In 2019 Kafle et al. [70] conducted a security analysis of the Google Nest and Philips Hue smart home platforms. They analyzed 761 smart home management apps from Google Play and Nest and found that 20.61% respectively 19.82% of the apps implemented insecure TLS certificate validation. Rahaman et al. [80] present the static analysis tool CryptoGuard, analyzed 6,181 Android apps in 2019 and found insecure TrustManager implementations in 25.30% of the apps. They conclude that Google Play's inspection safeguards are insufficient. Recently, in 2020, Weir et al. [86] performed an online survey with Google Play developers about their access to security experts and developer assurance techniques and analyzed their participants' apps using MalloDroid [54], CogniCrypt [72] and FlowDroid [40]. They found SSL issues in 70% of the apps. While previous work found that apps with vulnerable certificate validation logic have been published after 2016 in Google Play, our work is the first that conducts controlled experiments to investigate loopholes in Google Play's safeguard mechanism.

In 2020, Possemato and Fratantonio [79] analyzed the security of NSC settings in 16,332 apps. They find that many apps do not take full advantage of the NSC feature and allow insecure network protocols. In a root cause exploration they discover that developers copy & paste vulnerable settings from online resources and that several popular third-party libraries require developers to weaken their NSC settings. They conclude their work with a novel NSC extension that allows developers to include insecure libraries without weakening the security of the entire app. In contrast, our NSC analysis is based on a larger set of Android apps (99,212 instead of 16,332) and more detailed analyses (e.g. of certificate pinning issues and across app categories and download counts) and a manual analysis of 40 apps. Additionally, we perform a static code analysis on 15,000 apps and investigate the efficacy of Google Play's safeguards against vulnerable certificate validation logic in apps, providing a more complete picture of the current state of TLS security in Android apps.

Previous works suggested alternative approaches to custom TLS. As an example, in 2013, Fahl et al. [56] proposed a configuration approach to custom TLS behavior on Android, removing the need for developers to write any code. In a more recent approach in 2017, O'Neill et al. [76] introduced

TrustBase, a centralized approach to move TLS certificate validation to an OS service that intercepts all connections and enforces policies.

However, in contrast of providing another approach or research prototype for the custom certificate validation issue, our work focuses on the security analysis of NSC settings in Android apps and the efficacy of Google Play's safeguards.

4 NSC Adoption and Security

In this section, we illustrate the methodology of our NSC analyses and report their findings.

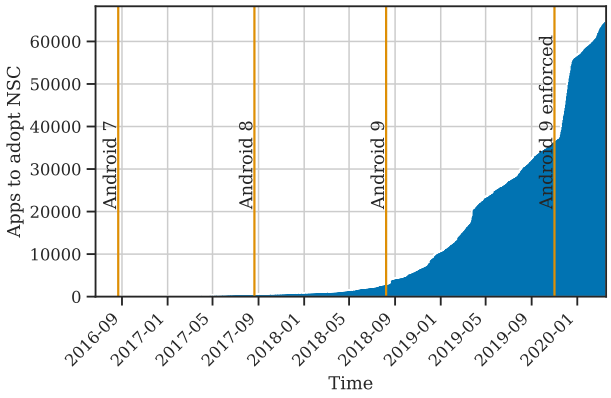

Figure 2: Adoption of NSC over time. The release of Android 9 had a significant contribution.

Body of Android Applications. We base our research on a body of 1,335,322 free Android apps available in Google Play that had received at least one update since August 2016 when Google introduced NSC (cf. Section 2) for Android 7. We downloaded the set of Android applications from Google Play using the unofficial Google Play protobuf API [13]. To grow the number of apps, we added apps from the "similar" apps section of an app's details website recursively. Overall, we collected apps between 2016/08/22 [2] and 2020/03/18 using Oltrogge et. al's Android app crawler [75].

Of the 1,335,322 free Android apps we analyzed, 99,212 implemented custom NSC settings. We used the OBFUSCAN tool [87] to detect obfuscation and excluded 2,812 (2,83%) obfuscated apps to improve our analysis quality. We conducted further analyses on the remaining 96,400 apps.[3]

Table 2 gives an overview of the target SDKs and custom NSC settings of the apps. Figure 2 illustrates the adoption of custom NSC settings: We see a significant increase with the release of Android 9. Similarly, we find that custom NSC settings are more frequently implemented in popular Android apps (cf. Figure 3). Even though Android rolled out NSC in September 2016 with Android 7 (cf. Table 1), widespread adoption was delayed until early 2019 (cf. Figure 2). This correlates with Android 9 introducing HTTPS as the default protocol for web requests in late 2018 (cf. Table 1).

Table 2: Body of Android apps: Total Apps vs. Apps with NSC

	Total Apps	Apps w. NSC
Target SDK		
< Android 7⋆	236,843	68
>= Android 7	1,098,479	96,332
>= Android 8	963,750	95,826
>= Android 9	565,910	88,854
Total	1,335,322	96,400

⋆ Though NSC is only supported for Android 7 and higher, apps with lower target SDKs can use backport-libraries (e.g., TrustKit. [25]) to implement NSC.

4.1 Security Analysis of Custom NSC Settings

Below, we analyze the security of custom NSC settings. Figure 3 illustrates our findings across app categories and download counts.

Measuring the Adoption of Custom NSC Settings: We begin with the detection of apps that include custom NSC settings. If an Android app contains custom NSC settings, a reference to the respective settings file is included in the `android:networkSecurityConfig` property of the AndroidManifest.xml. In cases of a missing reference, we check for the `android:usesCleartextTraffic` attribute to assess whether cleartext traffic is permitted for all network connections without using NSC. [6]

NSC Settings Analysis: Since we aim to gain insights on how NSC settings are used by developers [4], we extract and analyze all relevant information from the NSC files. First, we examine the high-level NSC features which are used by traversing the NSC file's XML document tree, starting with the root tag `<network-security-config>`.

NSC files with `<base-config>` elements include global options that affect connections for all hosts. The presence of `<domain-config>` elements indicates custom settings for specific hosts. Each `<domain-config>` element may include a set of custom settings for a list of hosts that can each be specified in a particular `<domain>` element. Table 3 provides an overview of the NSC elements and attributes we analyzed. The table also illustrates secure and insecure options for each attribute and explains why the given examples are insecure.

Overall, we analyzed 96,400 apps that included a NSC settings file. 95,940 of these implemented at least one custom NSC setting, while 460 apps contained an empty NSC file. Regarding app demographics, we find popular apps with more

[2] The release date of Android 7.

[3] We provide the full list of apps on this link.

[4] Cf. Section 2.1 for an overview of all possible NSC settings developers can configure.

Table 3: Security impact of NSC-settings for <base-config> and <domain-config>. A ✓denotes that an element or attribute can be used in the respective environment. The *secure* and *insecure* columns show which attribute values are considered (in)secure. The *reason* column gives a brief explanation why values are considered insecure.

base-config	domain-config	element	attribute	secure	insecure	reason
✓	✓		cleartextTrafficPermitted	-, false	true	allows HTTP without TLS
✓	✓	<certificates>	src	-, system	user	allows user trusted CAs
			overridePins	-, false	true	disables pinning
	✓	<pin>		always		adds a pinned certificate
	✓	<pin-set>	expiration	>10 days[a]	<10 days[a]	pinning not enforced after expiration date

[a] Recommendation as checked by Android LINT (cf. [22])

than 50,000 downloads to be more likely to include a custom NSC file (11-47%, cf. Figure 3). Below, we discuss the results of our analysis for the use of cleartext traffic, certificate pinning, custom CA certificates and debug configurations. Since apps may contain the same attributes in both base and domain specific environments, the numbers in the following sections may not always add up.

4.1.1 Cleartext Traffic

In this section, we analyze all apps that deviate from the standard and explicitly declare the `cleartextTrafficPermitted` flag in the NSC file. Since Android 9, cleartext traffic is disabled by default (cf. Table 1). Therefore, we distinguish apps that target Android 9 or higher from apps that target Android 8 or lower. We also distinguish apps with global settings from apps with domain-specific settings. In both `<base-config>` and `<domain-config>` environments we check for the presence of the `cleartextTrafficPermitted` flag. Depending on the environment, an application allows HTTP connections for all or only specific domains if this flag is set to *true*. Table 4 illustrates the frequency the use of the `cleartextTrafficPermitted` flag across different Android versions.

Altogether, we found 89,686 apps that used the `cleartextTrafficPermitted` flag. This element was present uniformly across all apps that used NSC settings in our dataset, with 89-97% of apps in all download categories using it. 88,769 (98.98%) used it to re-enable HTTP. However, only 4,093 (4.56%) apps used the flag to enforce HTTPS by setting `cleartextTrafficPermitted="false"`. In our dataset, 565,910 apps target Android 9 or higher. Of those, 84,060 (14.85%) – 57,123 globally and 34,246 for specific domains – allow HTTP connections, therefore downgrading the security for these applications. In 3,908 apps that target Android 9 or higher the `cleartextTrafficPermitted` flag is set to false, which has no security benefit, as HTTPS is enforced by default. These configurations have little impact in 4,804 apps that target Android versions lower than 9 as these can use HTTP without custom workarounds (cf. 2.1) or enforce HTTPS by explicitly setting the `cleartextTrafficPermitted="false"` flag. A small number of apps that target Android 8 or lower (185) does this and enforces HTTPS. We further check if the `android:usesCleartextTraffic` flag in the Manifest file was modified, which is the attribute used to enforce HTTPS traffic by default. Since this option is only applied if no NSC file was provided, the security downgrade to HTTP only affects apps without NSC. Within our sample, 196,155 apps explicitly set the flag. Of these, 177,391 apps have no NSC file. 174,369 apps target Android 9 or higher and use the `android:usesCleartextTraffic` flag to re-enable HTTP for all hosts.

Table 4: Frequency in our dataset and security impact of `cleartextTrafficPermitted` across Android versions

Target	**true**	**false**
>= Android 9	○	◐
Global	57,123	1,252
Domain Specific	34,246	2,712
Total	84,060	3,908
< Android 9	◐	●
Global	4,002	36
Domain Specific	826	151
Total	4,709	185
All Android Versions	88,769	4,093

○ Negative impact on security; ◐ No impact on security; ● Positive impact on security

In contrast, we found only 2,459 apps that use the flag to enforce HTTPS, of which 2,166 apply the setting as they do not utilize NSC. About twice as many apps allow HTTP for all domains (61,125) as opposed to only specific domains (35,072), while explicitly enforcing HTTPS is more common for specific domains (1,288 globally, 2,863 for domains). When HTTP is enabled for certain domains, we extract them and check whether HTTPS would also be available. Altogether, we found 84,060 apps that featured a HTTP down-

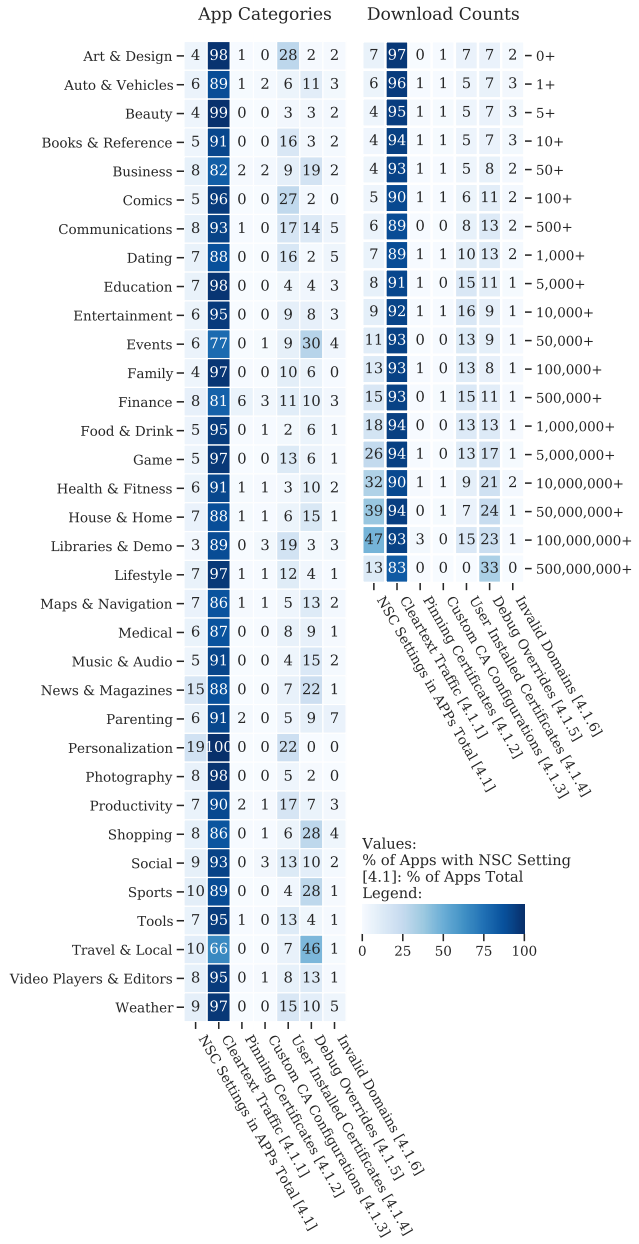

Figure 3: Distribution across the features we analyzed, and app categories, and download counts.

grade; this affected 24,653 distinct domains. We find valid HTTPS connections for 8,935 applications and argue that downgrading safe defaults was unnecessary. Table 8 in the appendix gives an overview of the most frequent domains for which we found downgrades.

Interestingly, the top domain values *127.0.0.1* and *localhost* seem to have no security impact. However, they might result from copy & paste from Facebook's cache proxy library that is used in many apps [20] or from debugging configurations developers use for testing.

Table 5 gives an overview of the most popular of these domains. We found 151 NSC configurations that upgraded to HTTPS; this concerned a total of 133 different domains.

Table 5: Top 10 domains where a HTTPS upgrade would be possible. All domains serve the same content over HTTP and HTTPS and most redirect from HTTP to HTTPS.

Apps	Domain Value	HTTPS Red.	Same Cont.
368	console-forum.net	✓	✓
294	securenetsystems.net	✓	✓
240	google.com	✓	✓
233	fineboost-loghub.ap-southeast-1.log.aliyuncs.com		✓
202	aff.bstatic.com	✓	✓
202	devel.tripwolf.com	✓	✓
202	www.tripwolf.com	✓	✓
190	competition-edge.com		✓
172	facebook.com	✓	✓
139	clients3.google.com	✓	✓

Table 10 in the appendix lists the most frequent domains for which connections were upgraded to HTTPS. Half of the entries contain invalid values such as URLs and resource IDs. They might stem from copy & paste events and have no security impact since domain values are expected.

It is hard to assess why developers chose HTTP over HTTPS. Reasons might include lack of knowledge, problems connecting via HTTPS or copying & pasting URLs from somewhere. Interestingly, in all cases where HTTPS would have been possible, the hosts serve the same content over HTTP and HTTPS and even redirect from HTTP to HTTPS in most cases. This is even more alarming: developers seem to suffer from a misconception and underestimate the threat of a MitMA in the presence of a redirect from HTTP to HTTPS.

4.1.2 Pinning Certificates

In this section, we report details for the certificate pinning analysis. We check if certificate pinning is used by searching for a `<pin-set>` element in `<domain-config>` elements.

Adoption. Overall, we found 663 apps that implement certificate pinning using NSC. We found 1,121 distinct pins for 2,781 distinct domains of which 998 are valid domains. Pinning was most common in the finance category (6%). This is in line with the most frequently pinned domains we found in Table 9, most of which belong to banking or mobile money apps.

Pinned Certificates. Our certificate analysis shows that 483 leaf certificates, 542 (intermediate) CA certificates and 289 root CA certificates were pinned. Table 7 in the appendix gives an overview of the most popular CA certificates. The majority of pinned CA certificates affected pre-installed system CAs. We extract the `<pin>` child tags and compare them with the certificates from domains we fetch certificate chains

for. To detect root CA pinning, we also match against pins generated from default Android system trust [4]. For 778 pinned domains we collected the complete certificate chain for the specified domains and analyzed it. We could not download all certificate chains due to connection problems or malformed domain names.

Backup Pins. The official Android documentation recommends the use of backup pins [19]. We found 566 apps that set a backup pin. In 47 cases, the pins were non-functional, e.g., empty strings were pinned. We discuss these cases in detail in section 4.1.6. For semantically correct pinning configurations, we find possible misconceptions regarding backup pins. First of all, the Android (Studio) LINT feature suggests to register two pins instead of one, but does not check for pin correctness or if both pins are equal [22]. We detect identical or non-functional backup pins by manual inspection and find cases containing sequences like *'AAAAAA...'*, *'BBBBBB...'* as prefix, or instances where only a single character is changed. While this is enough to address the LINT feature's warning, it does not enhance the security of the application. We also find that at least 12 applications used the empty pin hash produced by hashing an empty string encoded as Base64. This likely happens due to wrong usage of tools or lack of knowledge.

Pinning Expiration. The Android documentation suggests to set a pinning expiration date with the optional `expiration` parameter. After this date, pinning is no longer enforced, i.e., setting an expiration date may decrease security, but prevents an app from breaking when a certificate is replaced with a newer version [19]. Expiration values in the near future are critical from a security perspective as pinning would only be enforced for a short period. We read the respective element and found 130 apps that set a pinning expiration parameter. The mean expiration value was 947 days. Most apps had an expiration value set that had no negative impact on pinning security.

4.1.3 Custom CA Configurations

In both `<base-config>` and `<domain-config>` elements we check for `<trust-anchors>` elements which indicate modifications to the list of trusted root CAs to limit or add CAs.

We found custom CA configurations in 38,628 apps [5] (37,562 globally, 1,781 for domains).

759 apps distrusted all pre-installed CAs and added their own set of custom CA certificates (30 globally, 744 for domains). Furthermore, 123 apps restrict the list of pre-installed system CAs (14 globally, 112 for domains).

We further found 836 apps that added supplementary certificates (784 globally, 58 for domains). Table 12 in the appendix gives an overview of all added certificates and provides a summary of the most frequent custom CA certificates that apps used for production.

4.1.4 User-Installed Certificates

Based on the nested `<certificates>` element, we check if the value of the `src` property is set to *user* which enables trust for user-installed CA certificates. Compared to Android 7 default settings, enabling user-installed CA certificates is a security downgrade (cf. Section 2.1).

Out of 1,098,479 apps targeting Android 7 or higher, we found 8,606 apps that re-enable trust for user-installed certificates (8.67%) (8,001 globally, 707 for domains).

User-installed certificates are more common in popular apps. We found this issue more frequently in apps in the categories Art & Design (28%), Books & Reference (16%), Comics (27%) and Personalization (22%) (cf. Figure 3).

Since user-installed certificates increase the attack surface for MitMAs, developers are encouraged to use debug-overrides instead (cf. Section 4.1.5).

4.1.5 Debug Overrides

In this section, we present how app developers configured debugging settings.

Correct Use of Debug Overrides. `<debug-overrides>` can be used to debug secure network connections, e.g., using self-signed certificates or MitMA tools. The use of `<debug-overrides>` is a recommended security best practice, as these cannot be used in production code and apps with enabled debug flags cannot be published in Google Play. Overall, we found 10,085 apps with `<debug-overrides>`. Debug overrides were most popular among travel & local (46%) and event apps (30%), and generally among apps with higher download counts of 10,000,000 or more ((21-33%). We analyze their `<trust-anchors>` child elements for specific configurations of trusted roots. These can include user-installed certificates or bundled custom certificates which might be needed for MitMA proxies and other debugging purposes [10]. We found 318 apps that register custom certificates in `<debug-overrides>` (cf. Table 11 in the appendix). We detected 170 certificates of MitMA tools. 9,904 apps allow user-installed certificates in `<debug-overrides>`.

Mis-Use of Debug Overrides. Unfortunately, we also found several configurations outside the `<debug-overrides>` environment that we could unambiguously attribute to debugging purposes. 41 apps in our set use custom CA configurations to use MitMA certificates for debugging TLS connections. This was identified by observing the CA certificates' subject CN, in which popular MitMA proxy tools include the term **proxy**. For example, the *Charles Proxy* [9] MitMA proxy tool was the most popular in our dataset and included the substring *"Charles Proxy Custom Root Certificate"*. Contrary to

[5] We only discuss custom CA configuration in production code here. Custom CA configurations for debugging purposes are addressed in Section 4.1.5

<debug-override> configurations, these are used in production code and can therefore pose a security threat. Therefore, the Android documentation discourages their use [4]. While this list is not exhaustive, it shows that developers mis-use NSC settings for debugging purposes although NSC provides distinct debugging options.

4.1.6 Malformed NSC Files

In this section, we investigate faulty NSC files. We distinguish faulty configurations from configurations with syntax errors as they are simply ignored by Android and therefore do not negatively contribute to an app's security. Instead, we focus on configurations with ambiguous security settings resulting in confusing security implications.

Configurations with Flawed Domain Parameters. In 1,310 apps, we found <domain> configurations that contained an URL instead of a hostname, e.g., *http://example.com/* or *http://example.com/index.php* instead of *example.com*. In these cases, no error message is shown and the app compiles successfully. However, during app execution, such configurations are ignored and the <domain-config> setting becomes ineffective. We further identified 42 similar cases, where developers gave string resources (e.g., *@string/host*) instead of a hostnames. In 210 configurations, we found wildcard domain specifications (e.g. **.example.com*). These are also non-functional and therefore make the configurations ineffective.

Ambiguous Pinning Configurations. We analyzed our dataset for apps that include ambiguous pinning configurations, such as pins specified for the system-certificate with the overridePins flag, which overrides the pinning security benefits. We found 6, including two parental control apps and two that explicitly activate override pins for user-installed certificates, which developers registered as non-default trust anchors. Therefore, attackers can more easily mount MitMAs using social engineering. We further find all of these apps to be rather popular with more than 100,000 downloads. In 129 apps that pin specific domains we also found the permitClearTextTraffic="true" flag, which overrides pinning if HTTP is used instead of HTTPS.

Copy & Paste of Insecure Configurations. We investigate if apps contain NSC files that were copied & pasted from the Internet by manually inspecting common NSC snippets. We found applications that copy NSC snippets from information sources like library documentations, blog articles or Q&A threads [2,5,23]. We find NSC snippets in 496 apps that solve problems with an exception that requires HTTPS for specific network connections as HTTP is not sufficient. These snippets can be found on either StackOverflow [3] or in the MoPub app monetization documentation [11]. Overall, we find 1,609 applications that include a NSC snippet from the MoPub library documentation that instructs application developers to permit cleartext traffic globally [11] (cf. Listing 5 in the appendix). While the snippet permits cleartext traffic, it also restricts cleartext traffic for the domain *example.com*. For the cases we found, developers used the same code without making any changes. Similarly, we found 4 apps that use certificate pinning for the *datatheorem.com* or subdomains thereof. As these are related to Trustkit [25] and have no further effects, they are likely copied from the Trustkit demo application [26].

4.1.7 Impact of NSC on Android Ecosystem

Overall, NSC impacts app security on all levels of popularity. While most apps have less than 1,000 installs, there are numerous top apps with immense popularity. Within the most popular apps with more than a billion downloads, we find NSC to be mostly used to circumvent safe defaults, for example, to permit cleartext traffic in Android 9. This is the case for WhatsApp and several Google applications such as Youtube and GMail [39], all of which had more than five billion downloads. We further find a popular web browser that uses NSC to re-enable trust for user-installed certificates. We found all cases of misconfigurations and malformed NSC configurations we described in Section 4.1.6 in popular apps with more than one million downloads. Particularly interesting, we found one of the few cases where re-enabling trust for user-installed certificates leads to ineffective pinning. Similarly, we found copied & pasted code in apps with 100 million downloads. Overall, our findings suggest that the insecure use of NSC is not limited to amateur or unpopular apps.

4.1.8 Manual Analysis

Static analysis of NSC settings can show the potential for security problems for apps. However, the fact that NSC settings for insecure TLS certificate validation are present in an app's NSC or Manifest file does not necessarily mean that it is used or that sensitive information is passed along it. Even more detailed automated app analysis techniques cannot guarantee that all uses are correctly identified. Hence, we decided to conduct an in-depth manual investigation of affected apps. We aimed to find out what sort of information is actually sent over potentially insecure network connections. Therefore, we installed a set of apps that re-enabled HTTP cleartext traffic by installing the apps on an Android device and executing a passive MitMA against the apps. We focused on apps that re-enabled cleartext traffic since this vulnerability was widespread and is easy to exploit by a passive MitMA.

Therefore, we selected two sets of apps that re-enabled cleartext traffic for specific domains or globally:

Random apps. First, we selected and analyzed 20 random apps. We found 13 of them to use HTTP to transfer user data. In general, we found that affected apps use HTTP to transfer ad, tracking and debugging information including personally

identifiable information such as device identifiers. However, we also found a smart home app that allows users to remotely talk to their doorway devices and a school app that connects schools, parents and teachers. Both send sensitive account information including username and passwords from their users' devices to the service providers.

Privacy sensitive apps. Additionally, we analyzed 20 apps likely to handle sensitive data. [6] In this set of apps, eleven apps used HTTP. In all eleven apps we found login-related information, including usernames, emails, passwords, or passcodes. Similar to the random app set, we found one school for parents and a shopping app that send login credentials via HTTP.

In conclusion we find that in both sets more than half of the apps we tested manually used HTTP to transfer sensitive user data including login credentials.

5 Google Play Safeguards

In addition to NSC, Google Play changed their TLS policies and implemented new safeguards. In 2016, they announced to block new Android apps and updates that include insecure certificate validation code [67, 68].

In particular, Google announced to detect three implementations: TrustManagers vulnerable to attacks using invalid certificates [67], HostnameVerifiers vulnerable to malicious domains and hostnames [68], and WebViewClient.onReceivedSSLError implementations that do not appropriately handle HTTPS errors in a WebView [66]. To investigate root causes for the findings in previous work [70, 75, 80, 86] and the efficacy of these safeguards, we conducted multiple controlled experiments. We aimed to identify under which conditions Google Play still accepts apps with insecure certificate validation code. Therefore, we simulated a benign Android app developer who accidentally published vulnerable certificate validation code as part of their app. In each experiment, we included one or more vulnerable certificate validation implementations. After submitting each experiment to Google Play, the app went through the Google Play app review procedure. Once the verification process concluded, we checked for security alerts in the Google Play Console. [7]

Table 6 gives an overview of the four categories of experiments we performed: TrustManagers (TM), HostnameVerifiers (HV), WebViewClients (WV) and Libraries (LB). Libraries refer to insecure third party libraries we experimented with, trying to reproduce developer complaints we found online on GitHub [12, 17, 24, 27]. We also distinguish if the faulty code was reachable (R), hidden behind debug options (D), or unreachable (U).

Table 6: Details of our TLS security policy experiments.

Experiment	Reachability	Passed	Validation Logic
TrustManager			
TM-U	○	✓	No Validation at All
TM-R	●	✓	No Validation at All
TM-D	◐	✓	No Validation at All
TM-R-renamed	●	✓	No Validation at All, Renamed
TM-R-expired	●	✓	Cert Is Not Expired
TM-R-selfsigned	●	✓	Cert Is selfsigned and Not Expired
TM-R-chain	●	✓	Cert Has a Chain
TM-R-chainexpired	●	✓	Cert Has a Chain or Is Not Expired
HostnameVerifier			
HV-R	●	✓	No Validation at All
HV-D	◐	✓	No Validation at All, Debug switch
HV-R-global	●	✓	No Validation at All, Used by Default
HV-R-contains	●	✓	Verify Hostname Using "string.contains"
WebViewClient			
WV-R	●	X	always proceed
WV-D	◐	✓	always proceed, Debug switch
WV-wrapped	●	✓	always proceed, Depend on invariant condition
Library			
LB-U-acra	○	X	Acra with Insecure TM
LB-U-jsoup	○	✓	JSoup with Insecure TM and HV
LB-U-asynchttp	○	✓	async-http with insecure TM

● Always (R)eachable; ◐ Hidden Using a Debug Flag; ○ (U)nreachable
✓ App was accepted by Google Play; X App was blocked by Google Play

5.1 TrustManager Implementations

We started with investigating insecure TrustManager implementations [54, 58].

Empty TrustManager. First, we conducted experiments on an empty TrustManager implementation. Therefore, our test app used to download a file from a remote server. This was one of the most common insecure implementations [54, 58] and is frequently discussed in online Q&A forums [15, 16]. Given Google Play's announcement [67], this insecure implementation should be rejected. For full coverage, we tested multiple different empty TrustManager implementations: One that could be toggled with a debug flag (TM-D)[8], one that was always used (TM-R) and finally one where the TrustManager code was unreachable (TM-U). None of these implementations was blocked by Google Play. Additionally, we renamed the TM-R implementation to `TrustAllTrustManager` (TM-R-renamed) to match the most common TrustManager name reported by Fahl et al. [54]. This passed as well, which implies

[6] To identify these apps, we extracted static HTTP URL strings from app apks, tested their availability on the default HTTP port 80, and selected apps with URLs containing substrings such as 'login', 'register' and 'secure'.

[7] In case a vulnerable app was accepted, we removed it immediately to avoid that clueless users would install vulnerable software on their device. Given Google Play's download reports, no user installed one of our vulnerable apps.

[8] Some apps use such a flag for debugging purposes.

that the verification process employed by Google does not test for empty TrustManagers.

Non-Empty but Insecure TrustManager. Since not all insecure implementations reported in previous work [54] and discussed in online developer forums [29] are empty implementations, we extended the TrustManager experiments from above to also investigate non-empty but still insecure implementations. First, we implemented certificate validation logic that only tested for the server's certificate expiration date (TM-R-expired, TM-R-chainexpired, TM-R-selfsigned). Second, we tested an implementation that only checked whether the server sends a certificate chain (TM-R-chain). In both cases we did not implement secure certificate validation and only tested code that was always reachable. Again, Google Play accepted both vulnerable implementations.

5.2 HostnameVerifier Implementations

Our second set of experiments investigated the efficacy of Google Play's safeguards against insecure hostname verification in apps [54].

Always True Hostname Verification. We started with including HostnameVerifier implementations that accept any hostname for a certificate. We tested both, a reachable (HV-R) and a debugging implementation that was protected by a boolean debug flag (HV-D). These implementations turned off hostname verification. We further investigated an app that registered a global HostnameVerifier for all TLS connections by calling the static `setDefaultHostnameVerifier` method for the `HttpsURLConnection` class with the HV-R implementation (HV-R-global). Google Play accepted all vulnerable implementations.

Insufficient Hostname Verification. Next, we tested a HostnameVerifier implementation, which included code that did not always return *true* but only included insufficient hostname verification logic. As discussed in previous work [54], developers publish apps with implementations that only check for substring inclusion instead of testing the entire hostname. Hence, our experiment included a respective implementation (HV-R-contains). Again, this faulty implementation was accepted.

5.3 WebViewClient Implementations

The experiments in this section investigate Google Play's safeguard efficacy against insecure HTTPS error handling in `WebViewClient` implementations.

No Error Handling at All. First, we investigated HTTPS error handling logic that ignored certificate validation errors entirely and always proceeded with the TLS handshake. Similar to the experiments above, we tested vulnerable code that was always reachable (WV-R) and code that was hidden behind a debug flag (WV-D). Google Play detected WV-R and blocked the app from being published. However, the slightly more complex implementation WV-D passed without warning.

Obfuscated Error Handling. Motivated by previous work [74], we included an experiment that obfuscated insecure error handling. We tested error handling logic that hides the proceed call behind a boolean expression based on an invariant check (WV-wrapped). Again, this vulnerable implementation passed the Google Play checks.

5.4 Reproducing Complaints of Developers

Finally, we conducted a set of experiments to reproduce complaints of Android developers we found online [12, 17, 24, 27] concerning problems with specific Android libraries. We searched StackOverflow and GitHub issues for Google Play Console warning messages in the context of vulnerable certificate validation and found three vulnerable versions of popular android libraries (LB-U-acra, LB-U-jsoup, LB-U-asynchttp).

Acra 4.2.3. We aimed to reproduce the GitHub issue [27] in which a developer reports that the use of the Acra [7] library that provides application crash reports for Android in version 4.2.3 was rejected by Google Play on 2019/11/20 (LB-U-acra). We isolated and tested the vulnerable implementation and were able to confirm this issue as our app was blocked with this specific version of the library.

JSoup 1.11.1. In this experiment, we aimed to reproduce an error report for the JSoup [18] library for HTML parsing in version 1.11.1. Developers report that their apps were rejected because of a vulnerable TrustManager implementation [17, 24] (LB-U-jsoup). Similar to the experiment above, we used the exact same version of JSoup in our test app. However, we could not reproduce the error. Our app passed the Google Play safeguards successfully.

android-sync-http 1.4.9. Finally, we looked at the android-async-http [1] library providing interfaces for HTTP connections. This library included a vulnerable TrustManager in version 1.4.9 [12] (LB-U-asynchttp) and like previous experiments passed successfully without warnings.

5.5 Insecure Apps in Google Play

Motivated by the experiments above and previous work [70, 75, 80, 86], we replicated a study performed by Fahl et al. in 2012 [54]. However, we replaced the outdated MalloDroid tool [54] with the most currently published tool for vulnerable certificate validation logic in Android apps CryptoGuard [80] that was released in 2019. CryptoGuard can detect cryptographic vulnerabilities in general and vulnerable certificate validation code of both Java programs and Android apps. We picked a random set of 15,000 Android apps for the CryptoGuard analysis. Since CryptoGuard does not perform reachability analyses, we cannot tell whether vulnerable code is

actually executed. Using CryptoGuard reports we also cannot distinguish developer code from third party library code. Following Rahaman et. al [80] we terminated app processing after 10 minutes, therefore possibly skipping the analysis of more complex apps.

Overall, we found 2,232 (14.8%) apps with vulnerable HostnameVerifier and 5,202 (34.7%) apps with vulnerable TrustManager implementations. Most of the affected apps implemented both vulnerabilities, resulting in 5,511 (36.7%) vulnerable apps total.

Surprisingly, these results are in line with reports from Fahl et al. [54] and Georgiev et al. [58] and show that the Google Play security checks for TLS are inefficient.

6 Limitations

Our work has the following limitations:

Body of Android Apps. The Google Play crawler we used to download apps works on a best-effort basis. We seeded the crawler with a small list of popular free Google Play applications and recursively downloaded all available similar apps. Although we were able to find 1,335,322 free apps that have received an update after Android 7, we cannot guarantee that we were able to find all free Google Play applications. However, the behavior of our crawler is in line with previous work [37]. We also limited our analysis to free apps and ignored paid apps. Although we cannot generalize our findings to paid Android apps, this is also in line with previous Android security research [46, 48, 50–52, 54–57, 74, 75, 85]. We deployed the crawler at a university in Germany which resulted in 77,676 apps that we could not download due to geographic restrictions. Similarly, we could not download 264,249 apps that were e.g. removed from Google Play between crawling meta-data and download of APK files.

NSC Analysis. We identified 99,212 apps with custom NSC files. However, we could not analyze 2,812 of them due to obfuscation. As for the analysis of NSC files, we might be limited in our analysis of data related to HTTP(S) origins and certification data since we downloaded HTTPS certificates from Germany. Hence, the availability of HTTPS for certain websites, certificate chains and corresponding pins from certificates might differ from the respective results in other regions. In addition, since we analyze older versions of applications, servers could have changed their configurations over time which might not reflect the contents of NSC files anymore. Both limitations might apply in situations where certificate data is fetched in order to calculate pins we want to match against either trusted roots or pins specified in NSC files. Likewise, especially pins for backup that are not yet in use might not be matched by us as they might not (yet) be visible. As there is only one NSC file per application, there is no distinction between configuration related to the main application and libraries. This limitation may also apply for the static code analysis we conducted, which means that our analysis might not accommodate or might be limited to account for e.g. runtime behavior or reflection.

7 Discussion

In this section, we discuss key takeaways and the lessons we learned from our analysis of TLS certificate validation security in 1,335,322 free Android applications from Google Play. We discuss our analysis results and compare the state of certificate validation security in Android in 2020 with results reported in 2012 [54, 58].

We report positive as well as disappointing trends. Android deployed multiple measures in response to security vulnerabilities related to certificate validation. The measures include the introduction of NSC to support developers in implementing custom certificate validation logic, the default enforcement of HTTPS in apps targeting Android 9 or higher, and Google Play safeguards in 2016 and 2017 to prevent the publication of apps included insecure certificate validation code. While new Android apps benefit from new secure defaults (e.g., HTTPS by default), our results show the need for further security improvements. In the following, we will address the problems we found and discuss possible improvements.

Customization is Harmful. We find that usually, whenever developers configure NSC files manually to handle TLS certificate validation, security takes a hit. Our results mirror the 2012 results by Fahl et al. [54] and Georgiev et al. [58] that showed that custom certificate validation implementations in Android apps lead to vulnerabilities. In 2012, the underlying problem was insecure code that turns off certificate validation in 95% of apps with custom certificate validation code. Our results show that the problem persists in customized NSC files. Out of the 99,212 apps with custom NSC files that we were able to identify in our body of Android applications, 88,174 (88.87%) apps included configurations that downgrade security compared to default settings, mostly due to developers re-enabling HTTP traffic. Dramatically, we were able to show that this is usually unnecessary since the remote servers often supported HTTPS. Similar to the 2012 results, we were also able to see that developers still tend to roll out debug configurations in their production apps, unnecessarily leaving users at risk. We also show that 8.67% of the apps that include custom NSC settings allow user-installed CAs, which attackers can exploit in MitMAs [77]. This is in line with findings of Possemato and Fratantonio [79]. While they investigated a smaller app set, our findings supports theirs in several ways: First, we can confirm NSC's dominating use to re-enable cleartext traffic (cf. 4.1.1). We report similar findings regarding configurations for *127.0.0.1* (cf. Section 4.1.1) and copy paste behavior (cf. Section 4.1.6). Furthermore, their insights extend the investigations regarding the impact of vulnerable library use reported in our work, but clearly corroborate our

findings in a bigger picture. Likewise, we can support their proposals for extending NSC.

Pinning is Still an Issue. As early as 2012, 2013 and 2015, Fahl et al. [54, 56] and Oltrogge et al. [74] showed that only a small portion of developers implement certificate pinning. In developer interviews and surveys, they found that pinning is too complicated for most developers to implement and that the implementations are often faulty. Android has since simplified the use of certificate pinning, which has become much more straightforward via the configuration of NSC files as compared to the more complicated implementation via custom code, which was previously necessary. Our results show that, even though pinning should, in theory, have become more accessible, the rollout of NSC has not led to increased pinning use. Only 0.67% of the apps we investigated use NSC's pinning feature. The (non-)use of pinning seems to, therefore, not only be caused by the complexity of its implementation. Additionally, pinning seems to be a feature that is only interesting for a small minority of developers. Our findings confirm the results of Possemato and Fratantonio [79] on the low occurrence of pinning in NSC files and unintentional misconfigurations of pins that occurs across their sample (cf. Section 4.1.2 and 4.1.6).

NSC Implementation is Error-Prone. We were able to detect several faulty and insecure NSC configurations. Even though these are not responsible for a large number of vulnerabilities, they show systematic weaknesses in the current deployment of NSC. We were able to find apps that used URLs instead of domain names for domain-specific configurations. While this type of erroneous configuration does not prevent the app from working, it ignores the setting for the domain. Similar to our findings, Possemato and Fratantonio [79] report problematic domain usage, e.g. by developers using both the dummy domain *example.com* or invalid parameters for pins or domains (cf. Section 4.1.6). However, in addition, we discover cases in which URLs, regular expressions or other invalid strings are added instead of domains, all of which can lead to apps becoming less secure despite the use of pinning due to non-functional configurations.

We trace these misconfigurations back to insufficient documentation and lack of support for the Android Studio IDE. Android Studio only provides basic XML support for NSC files. There is limited support for tags and attributes (only limited support for misspelled or wrong tags or attributes (e.g., URLs instead of domains) or duplicates (e.g., for pins)). Android Studio does not support auto-completion for NSC. Available Android Studio support is based on LINTING checks [22] for NSC and dates back to 2016. Since then, there were no significant enhancements.

We corroborate findings that resemble vulnerabilities found in 2012, 2013, and 2015 by Fahl et al., Georgiev et al., and Oltrogge et al. Nguyen et al. [73] showed that better developer support in the IDE has the potential to lead to significant improvements to app security. Similar approaches for NSC seem promising.

Google Play Safeguards are Insufficient. Even though Google Play announced safeguards for vulnerable implementations of certificate validation logic in 2016 and 2017, and the ability of state of the art tools [41, 80] to identify the vulnerabilities we tested, our findings and previous work [70,75,80,86] suggest that Google Play's present deployment of these checks is insufficient. We were able to publish simple but vulnerable implementations of `TrustManager`, `HostnameVerifier`, and `WebViewClient` code to Google Play that, according to Google's announcements, should have been detected and prevented. In addition to our experiments with publishing insecure certificate validation, we were able to use static code analysis to show that a multitude of newly released apps still contains vulnerable implementations. While we could not pinpoint the exact technical realization of Google Play's certificate validation vulnerability detection safeguards, tools such as CryptoGuard [80] or LibScout [41] would have detected the vulnerable apps we tested. Hence, we recommend Google Play to consider the integration of state of the art vulnerability detection mechanisms to detect and block vulnerable apps in the future.

8 Conclusion

In this paper, we continued the long history of research efforts covering the state of (custom) TLS certificate validation in Android apps. While earlier studies focused on dangerous custom TLS code and proposals to prevent this, we focus on the on-going evolution of Android. New secure defaults lead to better security regarding HTTPS adoption as well as making MitMA harder to mount. At the same time, NSC, rather than accelerating wide-spread secure use of pinning, is mostly used for degradation of security in apps by undermining safe defaults. Also, we find that Google Play's safeguards intended to prevent vulnerable TLS implementations in apps being published do not work as expected. Overall, our results confirm that customization is often harmful to an application's security.

Acknowledgements

We thank the anonymous reviewers of this and an earlier revision of this paper, who have all contributed significantly; and particularly USENIX shepherd Professor Adwait Nadkarni of the College of William and Mary.

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972).

References

[1] An Asynchronous HTTP Library for Android. https://github.com/android-async-http/android-async-http (visited on 09/22/2020).

[2] Android 7.0 unable to capture https packets. https://www.cnblogs.com/0616--ataozhijia/p/9766682.html (visited on 09/22/2020).

[3] Android 8: Cleartext HTTP traffic not permitted. https://stackoverflow.com/questions/45940861/android-8-cleartext-http-traffic-not-permitted (visited on 09/22/2020).

[4] Android Root CAs. https://android.googlesource.com/platform/system/ca-certificates/+/master/files/ (visited on 09/22/2020).

[5] Android WebView setCertificate issues SSL problems. https://stackoverflow.com/questions/6511434/android-webview-setcertificate-issues-ssl-problems/57951506#57951506 (visited on 09/22/2020).

[6] <application> | Android Developers. https://developer.android.com/guide/topics/manifest/application-element#usesCleartextTraffic (visited on 09/22/2020).

[7] Application Crash Reports for Android. https://github.com/ACRA/acra (visited on 09/22/2020).

[8] CertificatePinner. https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html (visited on 09/22/2020).

[9] Charles Web Debugging Proxy • HTTP Monitor / HTTP Proxy. https://www.charlesproxy.com/ (visited on 09/22/2020).

[10] Debug your app | Android Developers. https://developer.android.com/studio/debug (visited on 09/22/2020).

[11] Get Started with the MoPub SDK for Android. https://developers.mopub.com/publishers/android/get-started/#step-4-add-a-network-security-configuration-file (visited on 09/22/2020).

[12] Google Play Blocker: Unsafe SSL TrustManager Defined #1260. https://github.com/android-async-http/android-async-http/issues/1260 (visited on 09/22/2020).

[13] Google play python API. https://github.com/NoMore201/googleplay-api (visited on 09/22/2020).

[14] HostnameVerifier. https://developer.android.com/reference/kotlin/javax/net/ssl/HostnameVerifier (visited on 09/22/2020).

[15] Java android - uplaud apk and google play security alert. https://stackoverflow.com/questions/43847629/java-android-uplaud-apk-and-google-play-security-alert (visited on 09/22/2020).

[16] Java android . Google play security alert for insecure TrustManager. https://stackoverflow.com/questions/43777599/java-android-google-play-security-alert-for-insecure-trustmanager (visited on 09/22/2020).

[17] JSoup Issue: TLS Certificate Bypassable, throws warnings #912. https://github.com/jhy/jsoup/issues/912 (visited on 09/22/2020).

[18] jsoup: Java HTML Parser, with best of DOM, CSS, and jquery. https://github.com/jhy/jsoup/ (visited on 09/22/2020).

[19] Network security configuration. https://developer.android.com/training/articles/security-config (visited on 09/22/2020).

[20] Network security configuration - Caching on Android 9. https://developers.facebook.com/docs/audience-network/android-network-security-config/ (visited on 09/22/2020).

[21] Network security configuration | Android Developers | Opt out of cleartext traffic. https://developer.android.com/training/articles/security-config#CleartextTrafficPermitted (visited on 09/22/2020).

[22] Network security configuration LINT Checks - NetworkSecurityConfigDetector. https://android.googlesource.com/platform/tools/base/+/6c94f47a39aafc2f2dbd85c5263075c7a16c9297/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks/NetworkSecurityConfigDetector.java (visited on 09/22/2020).

[23] SwipeCardView network_security_config.xml. https://github.com/Gxyong/SwipeCardView/blob/master/app/src/main/res/xml/network_security_config.xml (visited on 09/22/2020).

[24] TrustAllX509TrustManager issue #909. https://github.com/jhy/jsoup/issues/909 (visited on 09/22/2020).

[25] TrustKit-Android: Easy SSL pinning validation and reporting for Android. https://github.com/datatheorem/TrustKit-Android (visited on 09/22/2020).

[26] TrustKit-Android: Sample NSC file. https://github.com/datatheorem/TrustKit-Android/blob/master/app/src/main/res/xml/network_security_config.xml (visited on 09/22/2020) (visited on 09/22/2020).

[27] Unsafe implementation of X509TrustManager #374. https://github.com/ACRA/acra/issues/374 (visited on 09/22/2020).

[28] Upload failed You uploaded a debuggable APK. https://github.com/phonegap/build/issues/436 (visited on 09/22/2020).

[29] Use X509TrustManager for SSL in android. https://stackoverflow.com/questions/49650900/use-x509trustmanager-for-ssl-in-android (visited on 09/22/2020).

[30] WebViewClient onReceivedSslError. https://developer.android.com/reference/android/webkit/WebViewClient.html#onReceivedSslError(android.webkit.WebView,%20android.webkit.SslErrorHandler,%20android.net.http.SslError) (visited on 09/22/2020).

[31] WhatsApp Hack Attack Can Change Your Messages. https://www.forbes.com/sites/daveywinder/2019/08/07/whatsapp-hack-attack-changes-your-messages-and-facebook-doesnt-seem-to-care/ (visited on 09/22/2020).

[32] X509TrustManager. https://developer.android.com/reference/javax/net/ssl/X509TrustManager (visited on 09/22/2020).

[33] Android M and the war on cleartext traffic, 2015. https://koz.io/android-m-and-the-war-on-cleartext-traffic/ (visited on 09/22/2020).

[34] CVE-2016-2402. Available from MITRE, CVE-ID CVE-2016-2402., Feb. 3 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2402 (visited on 09/22/2020).

[35] ACAR, Y., BACKES, M., BUGIEL, S., FAHL, S., MCDANIEL, P., AND SMITH, M. Sok: Lessons learned from android security research for appified software platforms. In *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)* (2016), IEEE.

[36] ALLEN, C., AND DIERKS, T. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999. https://rfc-editor.org/rfc/rfc2246.txt (visited on 09/22/2020).

[37] ALLIX, K., BISSYANDÉ, T. F., KLEIN, J., AND LE TRAON, Y. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 468–471.

[38] AMOUR, L. S., AND PETULLO, W. M. Improving application security through TLS-library redesign. In *Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2015, pp. 75–94.

[39] ANDRÉ, C. GMail Android App Insecure Network Security Configuration, 2018. https://labs.integrity.pt/articles/Gmail-Android-app-insecure-Network-Security-Configuration/ (visited on 09/22/2020).

[40] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI'14)* (2014), ACM.

[41] BACKES, M., BUGIEL, S., AND DERR, E. 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), ACM, pp. 356–367.

[42] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security (CCS)* (2014), pp. 394–405.

[43] BOEYEN, S., SANTESSON, S., POLK, T., HOUSLEY, R., FARRELL, S., AND COOPER, D. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. https://rfc-editor.org/rfc/rfc5280.txt (visited on 09/22/2020).

[44] BRUBAKER, C. Changes to Trusted Certificate Authorities in Android Nougat, 2016. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html (visited on 09/22/2020).

[45] BRUBAKER, C. Protecting users with TLS by default in Android P, 04 2018. https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html (visited on 09/22/2020) (visited on 09/22/2020).

[46] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)* (2011), ACM.

[47] CHOTHIA, T., GARCIA, F. D., HEPPEL, C., AND STONE, C. M. Why banker bob (still) can't get tls right: A security analysis of tls in leading uk banking apps. In *Financial Cryptography and Data Security* (Cham, 2017), A. Kiayias, Ed., Springer International Publishing, pp. 579–597.

[48] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[49] CUNNINGHAM, E. Improving app security and performance on Google Play for years to come, 12 2017. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html (visited on 09/22/2020).

[50] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in android applications. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)* (2013), ACM.

[51] ENCK, W., GILBERT, P., CHUN, B. G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones.

[52] ENCK, W., OCTEAU, D., MCDANIEL, P. D., AND CHAUDHURI, S. A Study of Android Application Security. In *Proc. 20th Usenix Security Symposium (SEC'11)* (2011), USENIX Association.

[53] EVANS, C., PALMER, C., AND SLEEVI, R. Public Key Pinning Extension for HTTP. RFC 7469, Apr. 2015. https://rfc-editor.org/rfc/rfc7469.txt (visited on 09/22/2020).

[54] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[55] FAHL, S., HARBACH, M., OLTROGGE, M., MUDERS, T., AND SMITH, M. Hey, you, get off of my clipboard - On How Usability Trumps Security in Android Password Managers. In *Proc. 2013 Financial Cryptography and Data Security (FC'13)* (2013), Springer.

[56] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)* (2013), ACM.

[57] FISCHER, F., BÖTTINGER, K., XIAO, H., STRANSKY, C., ACAR, Y., BACKES, M., AND FAHL, S. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)* (2017), IEEE.

[58] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[59] GOOGLE. Android 6.0 Changes. https://developer.android.com/about/versions/marshmallow/android-6.0-changes (visited on 09/22/2020).

[60] GOOGLE. Android 7.0 for Developers. https://developer.android.com/about/versions/nougat/android-7.0 (visited on 09/22/2020).

[61] GOOGLE. Android 7.0 for Developers. https://developer.android.com/about/versions/oreo/android-8.0-changes (visited on 09/22/2020).

[62] GOOGLE. App security improvement program. https://developer.android.com/google/play/asi (visited on 09/22/2020).

[63] GOOGLE. Behavior changes: apps targeting API level 28+. https://developer.android.com/about/versions/pie/android-9.0-changes-28#framework-security-changes (visited on 09/22/2020).

[64] GOOGLE. Security Enhancements in Android 6.0. https://source.android.com/security/enhancements/enhancements60 (visited on 09/22/2020).

[65] GOOGLE. Upload App. https://support.google.com/googleplay/android-developer/answer/113469#targetsdk (visited on 09/22/2020).

[66] GOOGLE. How to address WebView SSL Error Handler alerts in your apps, 2016. https://support.google.com/faqs/answer/7071387 (visited on 09/22/2020).

[67] GOOGLE. How to fix apps containing an unsafe implementation of TrustManager, 2016. https://support.google.com/faqs/answer/6346016 (visited on 09/22/2020).

[68] GOOGLE. How to resolve Insecure HostnameVerifier, 2017. https://support.google.com/faqs/answer/7188426 (visited on 09/22/2020).

[69] HE, B., RASTOGI, V., CAO, Y., CHEN, Y., VENKATAKRISHNAN, V. N., YANG, R., AND ZHANG, Z. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 519–534.

[70] KAFLE, K., MORAN, K., MANANDHAR, S., NADKARNI, A., AND POSHYVANYK, D. A study of data store-based home automation. CODASPY'19, Association for Computing Machinery, p. 73–84.

[71] KOZYRAKIS, J. An examination of ineffective certificate pinning implementations, 2016. https://www.synopsys.com/blogs/software-security/ineffective-certificate-pinning-implementations/ (visited on 09/22/2020).

[72] KRÜGER, S., NADI, S., REIF, M., ALI, K., MEZINI, M., BODDEN, E., GÖPFERT, F., GÜNTHER, F., WEINERT, C., DEMMLER, D., AND KAMATH, R. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), ASE'17, IEEE Press, p. 931–936.

[73] NGUYEN, D. C., WERMKE, D., ACAR, Y., BACKES, M., WEIR, C., AND FAHL, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proc. 24th ACM Conference on Computer and Communication Security (CCS'17)* (2017), ACM.

[74] OLTROGGE, M., ACAR, Y., DECHAND, S., SMITH, M., AND FAHL, S. To Pin or Not to Pin - Helping App Developers Bullet Proof Their TLS Connections. In *Proc. 24th Usenix Security Symposium (SEC'15)* (2015), USENIX Association.

[75] OLTROGGE, M., DERR, E., STRANSKY, C., ACAR, Y., FAHL, S., ROSSOW, C., PELLEGRINO, G., BUGIEL, S., AND BACKES, M. The rise of the citizen developer: Assessing the security impact of online app generators. In *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 102–115.

[76] O'NEILL, M., HEIDBRINK, S., RUOTI, S., WHITEHEAD, J., BUNKER, D., DICKINSON, L., HENDERSHOT, T., REYNOLDS, J., SEAMONS, K., AND ZAPPALA, D. Trustbase: An architecture to repair and strengthen certificate-based authentication. In *USENIX Security Symposium* (2017).

[77] ONWUZURIKE, L., AND DE CRISTOFARO, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)* (2015), ACM, pp. 1–6.

[78] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (2014), The Internet Society.

[79] POSSEMATO, A., AND FRATANTONIO, Y. Towards HTTPS everywhere on android: We are not there yet. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 343–360.

[80] RAHAMAN, S., XIAO, Y., AFROSE, S., SHAON, F., TIAN, K., FRANTZ, M., KANTARCIOGLU, M., AND YAO, D. D. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. CCS'19, Association for Computing Machinery, p. 2455–2472.

[81] RAZAGHPANAH, A., NIAKI, A. A., VALLINA-RODRIGUEZ, N., SUNDARESAN, S., AMANN, J., AND GILL, P. Studying tls usage in android apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2017), CoNEXT'17, Association for Computing Machinery, p. 350–362.

[82] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018. https://rfc-editor.org/rfc/rfc8446.txt (visited on 09/22/2020).

[83] RESCORLA, E., AND DIERKS, T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008. https://rfc-editor.org/rfc/rfc5246.txt (visited on 09/22/2020).

[84] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps.

[85] TENDULKAR, V., AND ENCK, W. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities. In *Proceedings of the IEEE Mobile Security Technologies workshop (MoST)* (2014), IEEE.

[86] WEIR, C., HERMANN, B., AND FAHL, S. From needs to actions to secure apps? the effect of requirements and developer practices on app security. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 289–305.

[87] WERMKE, D., HUAMAN, N., ACAR, Y., REAVES, B., TRAYNOR, P., AND FAHL, S. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proc. 34th Annual Computer Security Applications Conference (ACSAC'18)* (2018), ACM.

A Appendix

Table 7: Top 10 Root CAs detected in pinning

Apps	CA[a]
44	CN=Amazon Root CA 1
39	CN=Go Daddy Root Certificate Authority - G2
24	CN=Starfield Services Root Certificate Authority - G2
22	CN=DigiCert High Assurance EV Root CA
22	CN=DigiCert Global Root CA
19	CN=DigiCert Global Root G2
17	CN=Entrust Root Certification Authority - G2
16	CN=GlobalSign Root CA
16	CN=Baltimore CyberTrust Root
16	CN=COMODO RSA Certification Authority

[a] We use the CAs' `CommonName` attribute for brevity here

Table 8: Top 10 Domains with HTTPS downgrade.

# Apps	HTTPS	Domain Value
11,689		127.0.0.1
4,290		localhost
740		10.0.2.2
449		localdev.cc
392		amazon-adsystem.com
376		virenter.com
366		10.0.3.2
366	✓	securenetsystems.net
293	✓	renweb.com
290	✓	getfitivity.com

✓ HTTPS would be possible

Table 9: Top 10 domains that were used with pinning.

Apps	Domain Value	Exp	Leaf	CA
29	ayers.com.hk	✓	✓	
36	subaio.com			
24	finopaymentbank.in			
23	webmobi.com			✓
12	api.app.olbisoft.de		✓	
12	cmtelematics.com		✓	
12	info.app.olbisoft.de		✓	
11	demo.pay2india.com			
11	gmail.com			✓
9	app.sociabble.com	✓	✓	

* We could not find the certificate for the given pinning value.

Table 10: Top 10 Domains with HTTPS upgrade

# Apps	Domain Value
76	cdn.example2.com
76	example.com
8	horaires-aeroports.appspot.com
7	ayers.com.hk
4	apis.appnxt.net
4	10.0.2.2
4	10.0.3.2
4	http://credu.com
4	http://el.multicampus.com
4	http://www.credu.com

Listing 1: Empty TrustManager - Accepts all certificates

```
@Override
public void checkServerTrusted(X509Certificate[]
    chain, String authType) throws
    CertificateException {
}
```

Table 11: Top custom certificates for debugging

# Apps	Certificate
170	/CN=CharlesProxyCustomRootCertificate
65	/C=RU/L=Novosibirsk/O=CFT/CN=dev-new.bankplus.ru
12	/C=DE/O=aktivkonzepte/CN=aktiv-konzepte
9	/C=SI/ST=Slovenija/L=Ljubljana/O=Omsoftd.o.o./OU=Primoz/CN=OmsoftCA/emailAddress=primoz@omsoft.si
9	/CN=ng_test_ca_2/C=SI/O=Halcom/OU=NG
9	/C=SI/L=Ljubljana/O=Halcomd.d./OU=Corporate/CN=ljvfep3.halcom.local/emailAddress=sysadmins@halcom.si
8	/C=SI/O=Halcomd.d./OU=servercertificates/CN=fep-r3.halcom.local/SN=halcom.local/GN=fep-r3
8	/C=US/O=GeoTrustInc./CN=RapidSSLSHA256CA
6	/OU=Createdbyhttp://www.fiddler2.com/O=DO_NOT_TRUST/CN=DO_NOT_TRUST_FiddlerRoot
4	/C=CA/ST=PrinceEdwardIsland/L=Charlottetown/O=silverorangeInc./CN=roble/emailAddress=sysadmin@silverorange.com

* Certificates for Charles Proxy are generated during setup and include individual user and device names. Therefore, we only used the prefix for aggregation.

Table 12: Top custom certificates for production

# Apps	Certificate
647	/C=US/ST=NY/L=NY/O=NarviiInc./OU=Aminoapps/CN=https://aminoapps.com//emailAddress=system@narvii.com
379	/CN=console-forum.net
174	/C=US/ST=CO/L=Denver/O=Zerista/CN=*.zerista.d.dm7.me/emailAddress=dushyanth@zerista.com
174	/C=US/ST=CO/L=Denver/O=Zerista/CN=*.zerista.k.dm7.me/emailAddress=dushyanth@zerista.com
89	/CN=*.zerista.io
21	/C=AU/ST=Some-State/O=InternetWidgitsPtyLtd/CN=*.zerista.d.dm7.me
21	/C=US/ST=Colorado/L=Denver/O=Zerista,Inc./OU=Dushyanth/CN=*.zerista.k.dm7.me/emailAddress=dushyanth@zerista.com
16	/C=US/O=DigiCertInc/OU=www.digicert.com/CN=RapidSSLRSACA2018
16	/CN=CharlesProxyCA(1Jul2019,MacBook-Pro-de-Toni.local)/OU=https://charlesproxy.com/ssl/O=XK72Ltd/L=Auckland/ST=Auckland/C=NZ
16	/CN=CharlesProxyCA(20Nov2019,Marc.local)/OU=https://charlesproxy.com/ssl/O=XK72Ltd/L=Auckland/ST=Auckland/C=NZ

Listing 2: Empty HostnameVerifier - Accepts all hostnames

```
@Override
public boolean verify(String host, SSLSession
    session) {
    return true;
}
```

Listing 3: NSC permitting HTTP traffic again

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
...
<base-config cleartextTrafficPermitted="true">
...
</base-config>
...
</network-security-config>
```

Listing 4: Reactivating trust for user-installed CAs

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
...
<base-config>
    <trust-anchors>
        <certificates src="system" />
        <certificates src="user" />
    </trust-anchors>
</base-config>
...
</network-security-config>
```

Listing 5: Insecure NSC Snippet from the Mopub Library

```
<?xml version="1.0" ?>
<network-security-config>
        ...
        <base-config cleartextTrafficPermitted="
            true">
                <trust-anchors>
                        <certificates src="system"
                            />
                </trust-anchors>
        </base-config>
        <domain-config cleartextTrafficPermitted="
            false">
                <domain includeSubdomains="true">
                    example.com</domain>
                <domain includeSubdomains="true">
                    cdn.example2.com</domain>
        </domain-config>
</network-security-config>
```