Saarland University

Department of Computer Science

TLS on Android – Evolution over the last decade

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Marten Oltrogge

Saarbrücken, 2021

Tag des Kolloquiums:	07.04.2022
Dekan:	Prof. Dr. Jürgen Steimle

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Thorsten Herfet
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Sascha Fahl
	Prof. Dr. Ben Hermann
	Dr. Katharina Krombholz
Akademischer Mitarbeiter:	Dr. Zhikun Zhang

Zusammenfassung

Mobile Geräte und mobile Plattformen sind omnipräsent. Android hat sich zum bedeutendsten mobilen Betriebssystem entwickelt und bietet Milliarden Benutzer:innen eine Plattform mit Millionen von Apps. Diese bieten zunehmend Lösungen für alltägliche Probleme und sind aus dem Alltag nicht mehr wegzudenken.
Mobile Apps arbeiten dazu mehr und mehr mit persönlichen sensiblen Daten, sodass ihr Datenverkehr ein attraktives Angriffsziel für Man-in-the-Middle-attacks (MitMAs) ist. Schutz gegen solche Angriffe bieten Protokolle wie Transport Layer Security (TLS) und Hypertext Transfer Protocol Secure (HTTPS), deren fehlerhafter Einsatz jedoch zu ebenso gravierenden Unsicherheiten führen kann. Zahlreiche Ereignisse und frühere Forschungsergebnisse haben diesbezüglich Schwachstellen in Android Apps gezeigt.
Diese Arbeit präsentiert eine Reihe von Forschungsbeiträgen, die sich mit der Sicherheit von Android befassen. Der Hauptfokus liegt dabei auf der Netzwerksicherheit von Android Apps. Hierbei untersucht diese Arbeit verschiedene Möglichkeiten zur Verbesserung der Netzwerksicherheit und deren Erfolg, wobei sie die Situation in Android auch mit der generellen Evolution von Netzwerksicherheit in Kontext setzt. Darüber hinaus schließt diese Arbeit mit einer Erhebung der aktuellen Situation und zeigt Möglichkeiten zur weiteren Verbesserung auf.

Abstract

Smart devices and mobile platforms are omnipresent. Android OS has evolved to become the most dominating mobile operating system on the market with billions of devices and a platform with millions of apps. Apps increasingly offer solutions to everyday problems and have become an indispensable part of people's daily life.
Due to this, mobile apps carry and handle more and more personal and privacy-sensitive data which also involves communication with backend or third party services. Due to this, their network traffic is an attractive target for Man-in-the-Middle-attacks (MitMAs). Protection against such attacks is provided by protocols such as Transport Layer Security (TLS) and Hypertext Transfer Protocol Secure (HTTPS). Incorrect use of these, however, can impose similar vulnerabilities lead to equally serious security issues. Numerous incidents and research efforts have featured such vulnerabilities in Android apps in this regard.
This thesis presents a line of research addressing security on Android with a main focus on the network security of Android apps. This work covers various approaches for improving network security on Android and investigates their efficacy as well as it puts findings in context with the general evolution of network security in a larger perspective. Finally, this work concludes with a survey of the current state of network security in Android apps and envisions directions for further improvement.

Background of this Dissertation

This dissertation is based on the papers mentioned in the following. I contributed to all papers as one of the main authors.
The idea for the first work is inspired by the findings of Fahl et al. [114, 115] on parts of the author's master thesis together with findings on the limits of the browser-based security mechanism and the Public Key Infrastructure (PKI). The author conducted the analysis and toolchain implementation for the work. The analysis toolchain extends Sascha Fahl's initial implementation of MalloDroid [205] based on Androguard [22]. The author and Sascha Fahl discussed the results and Sascha Fahl motivated the analysis of telemetry data provided by Zoner [312] which was then conducted by the author. Yasemin Acar and Sascha Fahl contributed in conducting the developer study and writing the paper.
The idea for the work on application generators originates from Sascha Fahl's initial proposal in 2015 that application that are automatically generated instead of being developed from scratch every time could have a positive amplification effect on ecosystem security. In an early stage of this work, Yasemin Acar, Karoline Busse, Sascha Fahl and Christian Stransky contributed in the initial collection of application generators. Together with Sascha Fahl and Christian Stransky the author developed and implemented an initial version of a Play Crawler to create a local repository of Android apps for enabling large scale analyses of the Android ecosystem. The further investigation, analysis and extension of the set of generators was conducted by the author as well as the feature collection and implementation of the classification. At the same time, Sven Bugiel, Erik Derr and Giancarlo Pellegrino also worked on a security analysis of a subset of Android application generators. Therefore, forces on this topic were pooled. While the author conducted the classification and security analysis, the other authors contributed in parts of the security analysis as well as the discussion and interpretation of results including. Yasemin Acar, Michael Backes, Sven Bugiel, Erik Derr, Sascha Fahl, Giancarlo Pellegrino, Christian Rossow and Christian Stransky contributed in writing the paper.
Alongside the work on the topic of application generators in 2016, the author already identified Network Security Configuration (NSC) as a relevant configuration option not yet covered by prior work. The author implemented analysis tooling for detection and analysis for usage of Network Security Configuration (NSC). A first iteration of this analysis resulted in only identifying a very sparse adoption of this feature in only a few dozens of Android apps. Later on, after finishing the previous topic, the author, again, intensified efforts in the analysis of NSC usage. The author conducted implementation, evaluation and analysis. The author discussed the findings with Sascha Fahl. Based on past research on and evolution of Android and Transport Layer Security (TLS) this discussion lead to the decision to extend this work with a focus on revisiting findings of Fahl et al. [114] with state-of-the-art tools using CryptoGuard [243] as well as a study on the efficacy of hardened Google Play policies aiming at relieving from the dramatic state of TLS in Android apps. The corresponding analyses and evaluations were conducted by the author. Apart from that, Yasemin Acar, Sabrina Amft, Sascha Fahl and Nicolas Huaman contributed in writing the paper.

[P1] 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)*. USENIX Association, Washington, D.C., USA, Aug. 2015, 239–254.

[P2] 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: *Proc. 39th IEEE Symposium on Security and Privacy (SP'18)*. IEEE, San Francisco, California, USA, May 2018, 634–647.

[P3] Oltrogge, M., Huaman, N., Amft, S., Acar, Y., Backes, M., and Fahl, S. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: *Proc. 30th Usenix Security Symposium (SEC'21)*. USENIX Association, Vancouver, B.C., Canada, Aug. 2021.

Further Contributions of the Author

The author was also able to contribute to the following papers.

[S1] Fahl, S., Harbach, M., Oltrogge, M., Muders, T., and Smith, M. Hey, You, Get Off of My Clipboard. In: *Proc. 2013 Financial Cryptography and Data Security (FC'13)*. Ed. by Sadeghi, A.-R. Vol. 7859. Springer, Okinawa, Japan, Apr. 2013, 144–161.

[S2] Huaman, N., Amft, S., Oltrogge, M., Acar, Y., and Fahl, S. They Would do Better if They Worked Together: The Case of Interaction Problems Between Password Managers and Websites. In: *Proc. 42nd IEEE Symposium on Security and Privacy (SP'21)*. Vol. 1. IEEE, San Francisco, California, USA, May 2021, 1367–1381.

Acknowledgments

Arriving at this point after an adventurous and ongoing journey lasting vastly one decade and being able to distill the essence of my findings in this thesis would not have been possible without the support of my advisors, mentors, co-authors, colleagues, friends, supporters and my family.

Over the course of my journey in research and my dissertation, I have been gifted to be accompanied by amazing people. I am deeply thankful for Michael Backes as my supervisor, mentor and advisor. He gave me the opportunity to work within the unique environment in the Information Security & Cryptography Group/CISPA making this journey possible and giving me all the freedom pursuing my research. Likewise, I am sincerely grateful to Sascha Fahl for starting this unique journey by introducing me early on already to academia allowing me to gain direct experience, accompanying me starting a decade ago and being an invaluable source of support for on my way not only as mentor, advisor and co-author. In similar fashion, I am deeply grateful to Yasemin Acar for always being there on this whole path starting with virtually taking me to my first scientific conference. In addition, I would like to thank my amazing colleagues at CISPA, Leibniz University Hannover and University of Bonn in alphabetical order, namely Sabrina Amft, Sven Bugiel, Sergej Dechant, Erik Derr, Nicolas Huaman, Giancarlo Pellegrino, Christian Rossow, Christian Stransky. I am deeply grateful to you as co-authors for the research papers this dissertation bases on and all the effort, experience and energy that you put into together with me helping make these brilliantly shine. Further, I am grateful to all the great colleagues at TeamUSEC as a source of discussion on and inspiration for exciting research topics. Even more, for watching over and managing my virtual computing environments, storing tons of apps potentially taking more storage space than anything else and transferring crawler data more than once.

I would also like to thank Matthew Smith and Gabriele von Voigt from the former Distributed Computing and Security research group (DCSec) in Hannover. At DCSec, I attended my first security lecture that initially piqued my interest in the field of security. I am furthermore grateful to the Administrative, Scientific Strategy and Support Staff at CISPA and Saarland University for support in terms of challenges or problems over the course of this adventurous journey towards but not limited to the final stages of my dissertation phase.

In regards to finalizing and defending this dissertation I would sincerely like to thank the members of my examination committee. Professor Ben Hermann and Dr. Katharina Krombholz for serving as additional examiners for my thesis as well as Professor Thorsten Herfet for chairing my defense and Dr. Zhikun Zhang for taking protocol, and for the interesting questions and interested discussion during my defense making the defense of my dissertation a unique and remarkable experience for me that I am exceedingly grateful for.

Finally, I would like to sincerely thank my mother for everything, my brother and his family – particularly to my sister in law for revising this work – as well as my aunt and her family for their incredible support, along with my friends and all people having supported me on my way. Without those, this journey would not have been possible.

Contents

1 Introduction **1**

2 Background **9**
2.1 Introduction . . . 11
2.2 Android Platform . . . 11
2.2.1 Architecture . . . 11
2.2.2 Mobile Ecosystems and Appified Platforms . . . 12
2.2.3 Google Play as a Mobile Ecosystem . . . 12
2.3 An Android Application Crawler . . . 13
2.3.1 Web Crawler . . . 13
2.3.2 Version Checker . . . 13
2.3.3 Details Fetcher . . . 14
2.3.4 App Downloader . . . 14
2.3.5 Limitations . . . 14
2.4 Transport Layer Security . . . 14
2.4.1 Default Certificate Validation . . . 15
2.4.2 Man-in-the-Middle-attacks . . . 15
2.4.3 Mitigation Strategies . . . 18
2.5 TLS on Android . . . 21

3 Related Work **23**
3.1 TLS on Android . . . 25
3.1.1 Insecure TLS certificate validation . . . 25
3.1.2 Alternative approaches to custom TLS . . . 27
3.2 Code Reuse in Android apps . . . 28
3.2.1 Library Detection . . . 28
3.2.2 App Generators . . . 30
3.3 Transport Layer Security . . . 31
3.4 App Market Analysis . . . 32
3.5 Overall security in Android Apps . . . 32

4 Developers: Exploring applicability of pinning **33**
4.1 Introduction and Contributions . . . 35
4.2 Background . . . 36
4.3 An Exit Strategy . . . 37
4.3.1 Pinning in Android Apps . . . 37

4.3.2 Status Quo . . . 37
4.4 Classification Strategy . . . 38
4.4.1 Possible Recommendations . . . 40
4.4.2 Classification Details . . . 40
4.4.3 Challenges . . . 41
4.5 Implementation Details . . . 43
4.5.1 Disassembly . . . 43
4.5.2 Relevant API Calls . . . 43
4.5.3 Program Slicing . . . 44
4.5.4 Decide on Validation Strategy . . . 45
4.5.5 Limitations . . . 46
4.6 Evaluation . . . 46
4.6.1 Library Code . . . 47
4.6.2 Custom Code . . . 48
4.6.3 Update Frequencies . . . 51
4.6.4 Discussion . . . 54
4.7 Developer Support . . . 55
4.7.1 Feedback . . . 55
4.7.2 Tool Support . . . 56
4.8 Limitations . . . 57
4.9 Conclusion . . . 58
4.10 Summary . . . 59

5 App Generators: Externalizing security decisions to 3rd Parties 63
5.1 Introduction and Contributions . . . 65
5.2 Overview of Mobile AppGens . . . 67
5.2.1 Standalone frameworks . . . 68
5.2.2 Online Services . . . 68
5.2.3 Developer-as-a-Service . . . 70
5.3 Online App Generators . . . 70
5.3.1 Classification . . . 72
5.3.2 Fingerprinting Application Generators . . . 73
5.3.3 Market Penetration . . . 74
5.3.4 OAG App Characterization . . . 76
5.4 Analysis Methodology . . . 77
5.4.1 Boilerplate App Model . . . 77
5.4.2 Security Audit . . . 80
5.5 OAG-specific Attack Vectors . . . 80
5.5.1 Application Reconfiguration Attacks . . . 81
5.5.2 Application Infrastructure Attacks . . . 82
5.6 Evaluating Known Security Issues . . . 84
5.6.1 Best-practice Permission Usage (P1–P3) . . . 84
5.6.2 Insecure Cryptographic API Usage (P4) . . . 87
5.6.3 Insecure WebViews (P5–P8) . . . 88
5.7 Discussion . . . 91

5.7.1 Citizen App Developers on the Rise 91
5.7.2 Pitfalls of the “One Size Fits All” AppGens’ Strategy 91
5.7.3 Amplification of Security Issues 92
5.7.4 Missed Opportunity for a Large-Scale Security Impact 93
5.8 Conclusion 93
5.9 Summary 94

6 Platform: Rolling out security as new safe default 97
6.1 Introduction and Contributions 99
6.2 Background on TLS and Android 100
6.2.1 Network Security Configuration 101
6.2.2 Google Play 103
6.3 NSC Adoption and Security 104
6.3.1 Security Analysis of Custom NSC Settings 105
6.4 Google Play Safeguards 115
6.4.1 TrustManager Implementations 115
6.4.2 HostnameVerifier Implementations 117
6.4.3 WebViewClient Implementations 117
6.4.4 Reproducing Complaints of Developers 117
6.4.5 Insecure Apps in Google Play 118
6.5 Limitations 119
6.6 Discussion 119
6.7 Conclusion 122
6.8 Summary 122

7 Conclusion 125
7.1 Summary 127
7.2 Future Work 130

A App Generators 157

B Network Security Configuration 161

List of Figures

4.1 Statistics and Classification Results for Apps. 52
4.2 Statistics and Classification Results for Connections. 52
4.3 Time span between release of new app version and installation. 54

5.1 Software development workflow with standalone frameworks. 67
5.2 Software development workflow with online services. 69
5.3 Software development workflow with Developer-as-a-Service. 70
5.4 Download stats for apps from the top 5 OAGs. 75
5.5 Distribution of OAG-generated apps by Google Play store categories. . . 76

6.1 NSC structure. 103
6.2 Adoption of NSC over time. 104
6.3 Distribution across analyzed features, categories, and download counts. . 106

List of Tables

4.1 Relevant API Calls. 42
4.2 Distribution of Network API Calls. 47
4.3 Top 10 Third Party Libraries. 48
4.4 Origins in Custom Code – Connections marked with ✓ can be pinned. . 49
4.5 Top 10 Remote Origins. 50
4.6 X.509 Certificates Statistics. 50
4.7 Pinning Statistics (Conservative). 50

5.1 Classification and fingerprinting of OAGs. 71
5.2 OAGs grouped by app model. 79
5.3 Categorization of attack vectors against OAG-generated apps 85

6.1 Chronological overview of TLS-related events in the history of Android: . 102
6.2 Body of Android apps: Total Apps vs. Apps with NSC 105
6.3 Security impact of NSC-settings 107
6.4 Frequency and security impact of `cleartextTrafficPermitted`. . . 109
6.5 Top 10 domains where a HTTPS upgrade would be possible. 109
6.6 Details of our TLS security policy experiments. 116

A.1 Classification and fingerprinting of SAF and DaaS generators. 159

B.1 Top 10 Root CAs detected in pinning 163
B.2 Top 10 Domains with HTTPS downgrade. 163
B.3 Top 10 domains that were used with pinning. 164
B.4 Top 10 Domains with HTTPS upgrade 164
B.5 Top custom certificates for debugging 164
B.6 Top custom certificates for production 165

List of Algorithms

4.1 The Classification Process. 41
4.2 StringBuilder Analysis 45

List of Code Listings

2.1 Sample Code for making an HTTPS request 21
2.2 Insecure TrustManager - Accepts all certificates if they are not expired . 22
B.1 Empty TrustManager - Accepts all certificates 165
B.2 Empty HostnameVerifier - Accepts all hostnames 165
B.3 NSC permitting HTTP traffic again 165
B.4 Reactivating trust for user-installed CAs 166
B.5 Insecure NSC Snippet from the Mopub Library 166

1
Introduction

For more than the last decade, the advent and evolution of smart devices has had huge impact on everyday life of users. From traditional mobile communication and telephony utilities, smart devices have evolved to become smart personal assistants that provide an expendable platform and ecosystem, making them indispensable companions for users' everyday life, equipped with apps for any use case. The Android platform introduced in 2007 is the most dominant platform with a market share of more than 87% [26]. All functionality of these devices is presented to the user through apps that make use of devices' rich capabilities that they expose to app developers via Application Programming Interfaces (APIs) (cf. 2.2). Alongside traditional telephony services, these include, for example, camera functionality for taking pictures or recording videos, location services, sensor capabilities, mass storage and manifold connectivity capabilities such as internet access, WiFi or Bluetooth. While, in the beginning, devices provided storage for address book and Text Messages, nowadays, devices feature storage capacity comparable to notebooks or desktop computers. This enables users to have all their personal data such as documents, photos and videos at their fingertips. In addition, next-generation mobile networks provide increasing bandwidth and shrinking latency which allows devices to be always connected to internet servers making it user data being in sync on all their devices. All this together with the rich set of APIs and device capabilities entailing a great amount of autonomy for apps for any use case. These conditions enabled mobile app platforms to emerge and grow. The increasing interest led to a trend of so-called appification which is best described with the slogan "There's an app for that" coined and later trademarked by Apple, the creator of Android's strongest competitor [158].

In this appified world, which ranges from apps for connecting people to work together collaboratively, connecting with friends and families in social networks, keeping track of news, mobile banking, shopping, entertainment, note taking, controlling smart home devices, or fitness and health tracking, up to even allowing to utilize devices' sensor and connectivity capability protect users themselves and others [279]. With so many tasks and use cases shifting to the appified mobile world follows a massive increase in privacy-sensitive data that is handled by apps. Furthermore, same as on the internet, privacy-sensitive data in apps has been increasingly shared online since the increasing availability and utilization of online shopping, online banking, or social networks. Therefore, network traffic of apps has become a vivid target for eavesdropping or tampering of network communication by attackers who manage to successfully mount Man-in-the-Middle-attacks (MitMAs) (cf. 2.4.2). For protection against MitMAs on the internet, protocols such as TLS and its predecessor Secure Sockets Layer (SSL) [1] for establishing encrypted network connections (cf. 2.4). Although essential and unneglectable for modern internet applications and mobile apps, handling sensitive data, building, maintaining, deploying, and operating secure systems employing TLS can be highly demanding and many-faceted topic. Transport Layer Security or topics related have therefore been hotly debated in the research community up to popular media. Over more than the last two decades, there have been numerous security incidents and insecurities with respect to TLS. Root causes are manifold and, among other things, organizational failures such

[1] We use SSL and its successor TLS as synonyms in this work.

as mischievous Certificate Authorities (CAs) (cf. Section 2.4.2.2), crypto-algorithmic insecurities, or flawed server configurations. Another crucial cornerstone lies in the client-side implementation of certificate validation logic (cf. 2.4.1). Here, the Android platform forms no exception.

In 2012, Fahl et al. [114] and Georgiev et al. [126] were the first to deal with widespread erroneous TLS implementations overriding default certificate validation logic in non-browser software (cf. Section 2.5). Since then, a large body of research on insecurities pertaining to custom TLS in Android apps has evolved [114, 126, 115, 94, 274, 65, 263, 229, 313, 160, 19, P1, 74, 298, 226, 224, 85, 122, 244, 69, P2, 15, 181, 243, 295, 299, 187, 52, 239, P3]. Here we can find efforts that analyze the state of custom TLS implementations and risks imposed in apps [114, 115, 126, 263, 229, 313, 160, 85, 122, 244, 69, P2, 15, 181, 243, 295, 299, 187, 52, 239, P3] as well as we also find efforts that propose enhancements in regard to the contemporary state of custom TLS and network security on Android [115, 94, 274, 65, 19, P1, 74, 298, 226, 224]. The goal of this dissertation is to contribute to this line of research. It will extend the state of research on TLS in the Android ecosystem in multiple dimensions.

Commonly, the dramatic state of TLS on Android has been attributed to developers being overwhelmed with security settings, complex APIs, lack of training, comprehensive documentation, limited tool support, and safe defaults. To each of these, there have been manifold proposals to improve security in prior works (cf. 3.1.2). In contrast, as the main focus of this dissertation, we measure the effect, applicability, and efficacy of postulated and deployed measures in regard to improving app security in terms of enhancing TLS. These include adoption of Hypertext Transfer Protocol Secure (HTTPS), custom TLS as well as the employment and implementation of pinning. We consider proposals on different levels and with different actors (Developers [P1, P3], Third Parties [P2], and the Android Platform [P3]) as follows:

- **Developers** Means that individual developers can utilize to enhance network security to limit the attack surface for MitMAs and guard against erroneous implementations [P1, P3].
- **Third Parties** Shift away the duty for secure app development from individual developers and entrusting third party generators to produce secure apps and facilitating enhance network security [P2].
- **Platform** Introducing safe defaults and safeguards for enforcing an effortless baseline state-of-the-art security level instead of requiring developers to do on their own [P3].

Developers. First, we explore the capabilities of individual developers to upgrade network security in their apps. For this, we present a large-scale analysis on the applicability of pinning. In that, we fill the gap of a missing scrutinization of pinning as a panacea for the permissive default trust model in Android apps. The default TLS

implementation in Android apps is adopted from web-browsers and trusts a huge list of CAs which has been subject to criticism as it makes apps suspicious for the *weakest link* property of the Public Key Infrastructure (PKI) (cf. 2.4.2, 2.4.2.2 and 2.5) [289, 234]. Pinning has often been intuitively proposed as a panacea for this issue [31, 209, 231, 115]. However, there has not yet been a formalization and systematic evaluation on the general applicability of pinning in prior research. Therefore, we will elaborate on this gap. We will further investigate the trade-off between the security enhancement of pinning and the effort required for maintenance. Pertaining to the applicability of pinning, our systematization bases on ownership and exclusivity of origin and API calls. Based on this formalization, we conduct a large-scale analysis that extracts relevant information from Android apps using static code analysis in order to derive whether pinning is applicable or not. In addition, to assess the effort required for maintenance, we investigate the update behaviour of Android apps. For this, we analyze an extensive set featuring telemetry data collected from real users' devices in order to calculate the propagation time needed to maintain embedded certification data used for pinning. Regarding the power of individual developers, our large-scale results will give important insights in hindsight to their theoretic ability to deploy pinning as a measure for increase network security. In addition, based on our formalization, we propose an interactive tool that assists developers in the decision process and implementation of pinning. Prior works [114, 126] already thoroughly investigated the disastrous state of custom TLS implementations and proposed measures to keep app developers from having to implement their own possibly erroneous solutions [115, 274]. These, however, require modification of the Android framework and do not support developers in the decision in regards to whether pinning is applicable.

Third Parties. Second, we investigate third parties and their power for having a positive impact on network and app security in the Android ecosystem. We present the first comprehensive classification and security analysis of commonly used Online Application Generators (OAGs) for Android. We show how to fingerprint uniquely generated apps to link them back to their generator and thereby quantify the market penetration of these OAGs in a large-scale analysis. A large record of prior research has repeatedly uncovered security flaws in Android apps [107, 106, 82, 108, 238, 224]. These are commonly attributed to developers who are often untrained and no security experts [10, 156, 11]. In this work, we will therefore focus on new development models where control and decision making for security related aspects is externalized to third parties. Media already dealt with the emergence of a new trend towards app generators for so-called *citizen developers* [125, 164, 123]. This denotes tools that allow the creation of apps with little or no programming expertise. With an increasing pervasiveness of such tools, we hypothesize the duty of software quality assurance to shift away from developers to generators. From this, we derive the assumption for an amplification effect in regard to ecosystem security. Depending on whether these app generators produce secure or insecure apps, we hypothesize app generators can have a huge impact on the security of the ecosystem as vulnerabilities in one generator would, by design, affect all apps of that generator. Thus, in the positive case, this can be seen as a chance to increase overall security of the ecosystem. Prior research, however, already hints

at insecure TLS implementations in apps built using a generator framework for web developers [115]. Hence, with respect to a lack of systematic research on this topic, it is crucial to assess the security impact for this class of apps and assess their impact. Our work aims at closing this gap. In our work, we will focus on Online Application Generators (OAGs) that allow for complete creation of Android apps without the need for any programming expertise, which means that the complete responsibility and control is given to the generator. After assessing their market penetration, we present a sophisticated in-depth security analysis of apps generated with the most widely used OAGs. Following a systematization of architectural principles of OAG services and apps generated by these, we conduct a security a security analysis focusing on state-of-the-art best practises on Android including TLS development (cf. 3.5). Even more, with a distinction on network security, we analyze new classes of attacks that are abetted by the architectural design of apps created by generators and their use of networking APIs. As for the amplification effect, due to the unique position of OAGs, based on the results from our analyses, we will theorize on and investigate their opportunity for a large-scale positive effect on security, especially pertaining to network security.

Platform. Third, in the light of the first discovery on the epidemic pervasiveness of insecure TLS implementations in Android apps [114, 126], we revisit this topic and pursue the evolution of TLS on the Android platform since then. To address the problems of insecure TLS, Google adopted proposals on countermeasures from seminal research for ongoing development of Android [114, 115, 126, 274]. We present a comprehensive in-depth effort to assess the efficacy of these measures. This includes the introduction of Network Security Configuration (NSC) as a configuration approach for custom TLS such as pinning without the need for custom TLS implementation similar to proposals from prior research [115, 274]. This also includes a mechanism to enforce HTTPS by configuration analogous to as postulated before [114]. Further countermeasures were, in contrast, introduced at the ecosystem-level in the form of new safe defaults for apps and encompassing new security policies and safeguards for blocking apps with malicious TLS implementations. To this end, we present three studies. In a large-scale analysis, we assess the adoption of NSC in Android apps over time and assess the corresponding impact on network security. We consider safeguards introduced by Google to block apps with vulnerable TLS implementations and measure their efficacy by conducting an in-depth penetration testing study. We replicate a 2012 study by Fahl et al. [114] with state-of-the-art techniques [243] to assess the current state of TLS implementations in Android.

For each of these perspectives and stakeholders, we will investigate their capabilities in regard to have a positive impact on network security in Android apps. While our work largely focuses on network security on the Android platform, we will also deal with topics pertaining to the evolution of TLS and HTTPS in general and browser software in particular (cf. 2.4) as the developments in these fields thereby also have substantial impact on the facilities for enhanced network security on Android apart from previously discussed stakeholders regarding app development. To this end, for each discussed perspective in this dissertation, we will align our results also to the evolution

of the TLS ecosystem and the browser landscape (cf. 2.4).

Outline of this thesis The remainder of this work is structured as follows. Chapter 2 will introduce the necessary concepts and provide background information on Android and TLS followed by a survey of related work in Chapter 3. In Chapter 4, we study the applicability of certificate pinning. In Chapter 5, we feature an analysis of the amplification effect of Online Application Generators (OAGs). We deal with the evolution of custom TLS on Android in Chapter 6 with an emphasis on introduced security measures and facilities for hardening network security and guarding against erroneous custom TLS implementations at a platform level. With Chapter 7, we conclude the findings of our work and envision future research directions.

2
Background

2.1 Introduction

This chapter introduces general background information relevant to the topics throughout this thesis. First, we will give a brief introduction to Android and its appified mobile platform (cf. 2.2). This will give a brief overview of Android's architecture and character as app platform, as well as it will describe to what extent this will be relevant for this thesis. We will also briefly describe the architecture of our app crawler that was used for analyses throughout this thesis (cf. 2.3). Finally, we will also give a short overview of TLS in general and TLS on Android in particular regarding aspects that are relevant in this work (cf. 2.4).

2.2 Android Platform

Android is an open source operating system for mobile and smart devices. First introduced in November 2007, it is mainly developed by Google and the Open Handheld Alliance, a consortium of mobile platform stakeholders such as mobile operators, handheld manufacturers or software companies. [230]

2.2.1 Architecture

The Android operating system consists of multiple layers. In the following, we give a brief overview. For more thorough and extensive insights, we refer to further resources in the Android documentation (cf. [236]).

2.2.1.1 Linux Kernel and Hardware Abstraction Layer

On the lowest level, Android vases on a modified Linux Kernel. From a security perspective, the Linux Kernel helps to realize several key requirements including access control, process isolation or Inter-process communication (IPC).

On top of that, a hardware abstraction layer (HAL) exposes hardware functionality such as Camera, Bluetooth, network access or telephony to upper layers. These are provided and exposed as native libraries.

2.2.1.2 Android Runtime and API

A key component of the Android operating system is the Android Runtime. The Android Runtime provides a managed runtime environment for Android apps that are compiled to Dalvik EXecutable (DEX) code. For Android apps, the Android Runtime exposes

a rich set of Java APIs to build upon. These APIs bring functionality for core app components such as Activities, Services, Content Providers or Receivers and the View System. Apart from these, the Android Runtime exposes a manifold set of other APIs for Android apps including e.g. internet access.

2.2.1.3 System Apps

At the top level, there are Android apps that run inside the Android Runtime. They contain all visible artifacts and expose all kinds of device functionality to device users. The Android system is equipped with a set of system apps featuring core functionality such as Short Message Service (SMS) messaging, calendar, contacts or web browsing.

2.2.2 Mobile Ecosystems and Appified Platforms

The previously described base system can be arbitrarily modified and extended. Device manufacturers can selectively pre-install apps and system services on devices. It is even possible for third party apps to replace system apps if not prohibited by device manufacturers. These apps can operate on the same APIs and are executed in the same manner on the Android Runtime as previously described for system apps. To this end, device manufacturers can choose to include selected apps and services as mandatory platform components.

2.2.3 Google Play as a Mobile Ecosystem

Most prominently here are the Google Mobile Services (GMS). These provide a set of apps to be pre-installed on devices including Google Play and Google Chrome. This makes devices take part in Google's mobile ecosystem for Android. Android apps can use additional libraries and platform services e.g. for messaging. Users can install third party apps from the app store Google Play.

While there are different platforms and app stores, in this thesis we focus on Google Play. It is the most prominent and widely deployed mobile platform for Android devices, with a market share of 87% [26]. A majority of prior research efforts on Android apps has focused on Google Play. Especially the coverage of vulnerable custom TLS implementations in Android apps has first been uncovered for popular apps on Google Play [114, 126].

Our work will extend this line of research. In this work, we will therefore focus on Google Play. All functionality of these devices is presented This also allows us to shed light on state of network security in Android from different perspectives and at different times. This will include support for individual developers as well as impact of automation and standardization on ecosystem security in regards to application generators. Finally, we

will also elaborate on the impact of the platform itself. Here, we will investigate security measures, and evolution in tools and safeguards to measure their efficacy in regard to how these help to address the present state and problems uncovered by research in Chapter 6.

2.3 An Android Application Crawler

To conduct our analyses for Chapter 5 and Chapter 6, we implemented a crawler effort for downloading large amounts of Android apps and their contemporary versions. The web crawler component of our approach for discovering new apps on Google Play similarly as described by Carbunar et al. [76]. Our work on implementing the crawler started in 2015. This is before the advent of AndroZoo [17] in 2016. In contrast to AndroZoo, our crawler's main goal is not detection of malware and apps from various app stores. Instead, our approach is limited to Google Play [148] and focuses on extracting metadata pertaining to Android apps, their version histories and corresponding Android Package (APK) files for further analysis. The crawler consists of four different components. A web crawler for discovering new apps. A version checker for discovering new versions of known apps. A details fetcher to fetch app details and version information. An app downloader to download APKs for discovered versions. While the first component is realized as a common web crawler in Python, the other components make use of an unofficial implementation of the Google Play API [150]. All apps tracked by the crawling facilities are regularly revisited to keep information up to date.

2.3.1 Web Crawler

The web crawler recursively traverses the Google Play website. It resembles Carbunar et al.'s [76] approach and analyzes app description pages. From these, metadata such as application name, category, version name, author, last update, rating are extracted. Android apps on Google Play can be uniquely identified by their package name that is encoded in the app page's Uniform Resource Locator (URL). Extracted information is written to a database. To discover new apps, the crawler follows the links to related apps embedded in app pages. The apps to crawl are hold in a job queue. Initially, the crawler's queue is seeded with a list of popular apps. From this seed, the Google Play Store website has been subsequently crawled recursively. Whenever the crawler encounters a new app, it is added to the job queue. In addition, the app is added to the version checker's queue.

2.3.2 Version Checker

The unofficial Play API [150] provides an endpoint for bulk requests for metadata. Although, the response does not include the full metadata, it includes enough data

to detect new versions for 1,000 apps at once. For detection of new versions, the versionCode and date of last update is compared with the metadata already stored in the database. If a new version is detected, the app is added to the details crawler's job queue. Apps are regularly revisited for detection of new versions.

2.3.3 Details Fetcher

The details fetcher loads current metadata using the unofficial Play API [150]. Fetched details for new app versions are written to a database. Alongside, the tuple (app, versionCode) is added to the app downloader's job queue.

2.3.4 App Downloader

The app downloader component downloads APKs for apps based on their versionCode and writes them to disks. Moreover, after having downloaded an APK it can be added to further worker queues like e.g. for detection and analysis of NSC usage as needed for analysis in Chapter 6.

2.3.5 Limitations

The Google Play crawler works on a best-effort basis. We cannot guarantee that we are able to find all free Google Play applications. However, the behavior of our crawler is in line with previous work [17]. 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 [107, 106, 82, 108, 115, 274, 94, 114, 122, S1]. We deployed the crawler at a university in Germany, which might limit the amount of apps accessible due to geographic restrictions. Similarly, we might not be able to download apps that were meanwhile removed from Google Play between crawling metadata and download of APK files.

2.4 Transport Layer Security

For deployment of secure networking, TLS [101] is the most widely used protocol for secure communication between client and server. It allows to establish secure connections and provides *authentication*, *confidentiality* and *integrity*. For the first property, to establish secure TLS connections, *certificate validation* is important in order for communication partners to authenticate remote endpoints. Therefore, Android and most other non-browser software adopted the in-browser certificate validation strategy, i.e. applications trust a predefined set of *root certificate authorities* (root CAs) of trusted

third parties (TTP) that implement a PKI. This makes it possible to establish trust between unknown entities without prior knowledge. This is the common situation for client software like browsers when connecting to arbitrary remote servers using HTTPS employing TLS. Therefore, these TTPs are used to convey trust. When attempting to establish a secure connection, the server provides the client with a *certificate chain*.

2.4.1 Default Certificate Validation

Certificate chain validation works as follows:

Chain of Trust. Based on the certificate chain sent by the server, the client tries to build the *chain of trust* beginning at the server's leaf certificate up to one of the root CA certificates that are trusted by the client. Every certificate in the chain is checked for validity – i.e. it is not expired and it is signed by its immediate successor in the chain. The second last certificate in the chain must be signed by one of the root CAs installed on the user's device [68]; the last certificate in the chain belongs to the signing root CA.

Hostname Verification. A certificate is bound to a certain identity – in this case a particular hostname or a set of hostnames. During *hostname verification*, a client verifies this identity by checking whether the hostname can be matched to one of the identities (i.e. CommonName, SubjectAltName [248]) for which the certificate was issued.

Further checks and connection negotiation. The complete certificate validation process is more complex and includes further checks. In addition, after successful *authentication*, later stages of the TLS handshake aim to negotiate cryptographic properties to set up a secure communication channel providing *confidentiality* and *integrity*. These parts, however, are out of scope of this dissertation. For more in-depth information, we refer to [68] and related work (cf. 3.3).

2.4.2 Man-in-the-Middle-attacks

Adversaries stealing or manipulating sensitive information exchanged via network connections is a severe danger that TLS aims to protect against. In a Man-in-the-Middle-attack (MitMA), the attacker is in a position to intercept network communication. A passive MitMA can only eavesdrop on the communication, while an active MitMA can also tamper with the communication. Correctly configured TLS together with proper certificate validation is fundamentally capable of preventing both passive and active attackers from executing their attacks. There are multiple causes that can facilitate mounting of MitMAs. In this work, we will on the following:

Use of cleartext communication: The use of insecure protocols such as Hypertext Transfer Protocol (HTTP) and absence of TLS does not provide any protection against MitMAs. There is a long history pertaining to the adoption of HTTPS as a basic mean for mitigation. Section 2.4.2.1 will give a brief overview.

Mis-issued or mis-used certificates: Certificates play a central role for authentication during connection establishment. Due to having to rely on TTPs certificates issued by these trusted CAs to serve as proof of identity for a server to authenticate. However, if a CA falsely issues a legitimate certificate to an illegitimate entity, this can be used for mounting MitMAs. Breaches or cases of rogue CAs are a major problem. Section 2.4.2.2 will give more insights on this.

Erroneous certificate validation logic: Misconfiguration of server software and erroneous implementation of the certificate validation process in client software can lead to severe vulnerabilities, making mounting of MitMAs even more feasible. This is especially a problem in non-browser code such as Android apps. Here, developers' erroneous custom TLS code can render apps vulnerable to MitMAs accepting arbitrary certificates. We will further investigate this in Section 2.5 and throughout this thesis.

2.4.2.1 Adoption of HTTPS

Although already introduced in 1995, SSL was only sparsely used by website operators in the context of banking and credit card companies. Here, we give a brief overview on the history of TLS and HTTPS adoption in the browser landscape based on broader overviews that we refer to for more details (e.g. [182] and [276]). In the following years, HTTP remains the standard protocol for websites with HTTPS usage being limited to login forms while the rest of a website remains unprotected. The advent of Firesheep [75] demonstrated the insecurity of this practice by making session hijacking attacks feasible. This imposed a growing attack surface, especially due to internet access increasingly shifting from wired to wireless connections including public WiFis. Wireless internet access especially via public WiFi facilitated mounting of MitMAs for attackers. Major website operators since then started to offer HTTPS as an option for better protection. Browser extensions such as HTTPSEverywhere [171] helped users to automatically enforce the use of HTTPS if possible. Another important event leading to increasing adoption of HTTPS can be found in the Snowden revelations [157] that disclosed massive global surveillance efforts of intelligence agencies. These lead to a change of perspective as a driving force to propose HTTPS to not only being used for the login process or other security related tasks but actually for the complete traffic of websites as common practice. Major players like Facebook [111] finalized their transition to HTTPS by switching from the previous option to use HTTPS to using HTTPS by default while Google [240] having already switched earlier. Still, for website operators deployment and use of HTTPS remained cumbersome and costly as certificates needed to be purchased from CA companies. Correct deployment has been a challenging task for administrators as found by prior research [112, 186]. For removing these barriers

hindering a steeper increase in adoption of HTTPS, Let's Encrypt was founded as an alternative free CA in 2014 [196]. Fully publicly available starting April 2016 [193] , Let's Encrypt provides free certificates for website operators as well as advanced means to ease and automate the certification and deployment process [63]. Since then, browser vendors have continuously worked on encouraging website owners to adopt HTTPS and protect privacy of users. These efforts include making advanced modern Web-APIs like *WebWorkers* [288] or privacy-sensitive APIs like webcam and microphone access [210] only for available within secure browsing contexts (i.e. websites loaded securely) [256]. Further actions include special HTTP headers due to HTTP Strict Transport Security (HSTS) [165] for website operators to enforce HTTPS and protect against *SSL-Stripping* attacks [206, 257] as well as security indicators in browsers labelling websites loaded via HTTP as insecure or denial of mixed-content [88] with the long term eventual goal to overcome usage of HTTP traffic completely [86, 121]. On this path, the adoption of HTTPS has steadily increased from below 30% in 2014 and approx. 50% in 2017 to more than 80% in 2020/2021 based on Let's Encrypt statistics [195, 194] via telemetry data from Mozilla [264] and Google Transparency Report [170]. In parallel, a similar path for transformation from HTTP to HTTPS and related challenges have to be taken by app developers on Android. In this work, we will also cover and investigate the risks and problems regarding the continuing use of HTTP in Android apps as well as platform strategies for enforcing the adoption of HTTPS along with assessment of their efficacy (cf. Chapter 6).

2.4.2.2 Rogue CAs

While the adoption of HTTPS already constitutes a key security measure for protection against MitMAs, this opens new attack vectors to be addressed. For assuring the authenticity, clients completely depend on CAs operating correctly as trust for a server is bound to certificates issued by these authorities. Contemporary operating systems and browsers usually trust more than 100 CAs [234]. Sections 2.4.1 and 2.4.2 have already denoted both, the dependency on CAs as well as the massive security impact of CAs malfunctioning which lead to mis-issuance of certificates. One misbehaving CA could potentially mis-issue any illegitimate certificate to an adversary putting the whole TLS ecosystem at risk making the trust in CAs a weakest link property of the CA-PKI approach. Root causes for such cases are manifold and include insufficient or erroneous implementation of validation processes as well as human errors. There is a long history of such incidents. For an overview, we refer to comprehensive compilations such as [77] and [266] that feature a manifold list of CA breaches and failures affecting several major CAs including Thawte in 2008 [314], StartCom in 2008 [314], CertStar in 2008 [5], Comodo in 2011 [93, 92], DigiNotar in 2011 [290, 225], TurkTrust in 2011 [147], TrustWave in 2012 [284], NICCA in 2014 [190], CNNIC in 2015 [191], WoSign in 2015 [173, 253], Symantec in 2015 [260, 55], StartCom in 2016 [268], Comodo in 2016 [172], Symantec in 2016 [213], Certinomis in 2018 [4]. For accommodation, the CA/Browser Forum (CA/B) introduced baseline requirements [64] as minimal security standards for the operation of CAs and vendors. Furthermore, these events and incidents influenced the

ongoing evolution of the TLS ecosystem and have inspired and motivated research and development strategies for mitigation. In the following, we will cover some that are relevant to topics in this thesis.

2.4.3 Mitigation Strategies

To tackle the security problems related to the CA-PKI approach, various enhancements have been proposed. Over the years, alternative certificate validation strategies have come up [192, 301, 207, 208, 166, 231]. While they all provide approaches to check a certificate chain's validity, our work in this dissertation focuses on pinning [231] as one of the most recommended alternative strategies for non-browser software due to the need for prior knowledge on origins to connect to. In the browser landscape, for Chrome, Google already made use of *pinning* for its own domains in 2011 [87]. Section 2.4.3.1 will introduce to pinning followed by the broader approach of pinning called Trust on First Use (TOFU) in Section 2.4.3.2. In addition, there have been several developments in the CA-PKI for enhancing security especially in browsers but relevant for the whole TLS ecosystem (cf. Section 2.4.3.3).

2.4.3.1 Pinning

Pinning is a notion of certificate validation that uses existing knowledge about the network origin or the certification data presented to protect the TLS connection [231]. Required parameters to be pinned need to be available in an application before the TLS handshake happens. This aims at decoupling the authentication process from the need for relying on CAs as TTP and thus protecting against breaches. In Chapter 4 we will elaborate on the applicability of this strategy in Android apps. Pinning can be achieved based on different parameters (e.g. public key, whole certificate). Pinning can also be applied to different subjects:

Leaf Pinning. Leaf pinning includes pinning a single leaf certificate or its public key or a set of leaf certificates or their public keys and is the most rigorous way of pinning.

CA Pinning. Certificate authority pinning effectively limits the number of trusted certificate authorities and allows for more flexible reconfiguration of deployed TLS certificates. This includes both intermediate and root CAs.

2.4.3.2 Trust on First Use (TOFU)

Another notion of pinning is TOFU . Instead of knowing the information to be pinned in advance, the first certificate (leaf or CA) seen for a TLS connection is stored locally on the client side and used to validate certificates for further connections. Hence,

TOFU can be seen as a mix of conventional pinning (cf. Section 2.4.3.1) and the default validation strategy. TOFU is used to secure SSH connections and is an opt-in feature for HTTPS web servers for which it is called HTTP Public Key Pinning (HPKP) (cf. Section 2.4.3.3) [110, 185].

2.4.3.3 Evolution of the CA-PKI ecosystem

Certificate Revocation. To address mischievous acting CAs browsers' certificate validation logic includes further checks to check for *revocation* of certificates before opening an authenticated connection. Via Certificate Revocation List (CRL) [68] or Online Certificate Status Protocol (OCSP) [251] CAs can revoke misused certificates and notify browsers. However, checking and managing certificate status turned out to be accompanied by massive performance penalties and organizational overheads as well as security impacts due to delays and risk of Denial of Service (DoS) and blocking attacks [189]. As a consequence, in 2012, Chrome stopped checking for certificate status and instead introduced CRLSets (CRLSets) as an alternative approach. CRLSets decouples certificate status checking from opening connections by pushing information for *revocation* using an update mechanism [189]. While addressing some shortcomings of *revocation*, subsequent developments including Certificate Transparency (CT) and HPKP aim for alleviation and better protection against falsely issued certificates.

HTTP Public Key Pinning. Analogous to previous efforts in Google Chrome to utilize a notion of pinning (cf. Section 2.4.3.1) by embedding hardcoded pins for selected domains in 2011 [87], HTTP Public Key Pinning (HPKP) is formed as a new web standard and as a mean for server operators to make use of pinning to protect against forged certificates. Chrome Version 46 adds support for HPKP in 2015 [110, 267]. HPKP operates in a TOFU manner (cf. Section 2.4.3.2). Thus, storing pins on first encounter and only trust these for a specified time frame. For case of mismatch, server operators can opt to either only be notified or to halt the connection as protection against MitMAs. However, following its introduction, HPKP turned out to be cumbersome complex to maintain [162]. When not configured correctly, it can even cause more risks than it solved. Lack of setting proper backup pins can lead to sites becoming inaccessible when the servers' certificate changes making it possible impossible to issue a certificates for the respective domain as only the old certificates pin is trusted. In the same manner, this can even be used as an attack vector if an attacker gains access to a web server's configuration. Due to these problems and its low adoption [161, 119] , HPKP got deprecated in Chrome Version 67 in 2018 [174] and finally removed with Chrome Version 72 in 2019 [247, 163].

Certificate Transparency. Developed by Google, Certificate Transparency (CT) [90, 192] is launched in 2013 with the first log as an alternative approach for detecting and reacting to mis-issued certificates. CT aims at certificates issued by CAs to be added to public append-only logs. This way, CT serves to bring accountability to the CA-PKI trust model by making CAs' misconduct public. Monitors serve to audit for suspicious

behaviour making it impossible to hide misbehaviour such as issuing of certificates to illegitimate entities. After requiring Extended Validation (EV) certificates to be added to a CT log in 2015, in 2018, Chrome, along with other browser vendors, started to require all certificates to be added to logs for being trusted by browsers [90].

DNS Certification Authority Authorization. In 2013, DNS Certification Authority Authorization (CAA) is introduced as another measure against the problem of mis-issuance of certificates and thus the risk for MitMAs [159]. In contrast to HPKP, CAA targets CAs. With CAA, website operators can indicate in their Domain Name System (DNS) record which CAs are allowed to issue certificate for the corresponding domain. Other CAs honoring CAA are consequently expected not to (mis-)issue certificates for a given domain that adversaries could use to mount MitMAs. Thus, CAA aims to function as precaution against failures in the CA-PKI ecosystem. In 2017, the CA/B voted for CAA to honor [61]. In their study in 2017, however, Scheitle et al. [252] find that CAs's implementations of CAA are error-prone. Likewise in 2021, the adoption of CAA by domain owners is still low with 8.7% which, therefore, limits its effect so far [241].

Certificate Lifetime. With CT already striving to make CAs' failures in adhering to baseline requirements visible to the public and thus allowing for faster awareness and reaction, Google proposed to reduce the lifetime of a certificate in 2019 [62]. Shorter lifetimes for certificate help reducing the risk of exposure to the ecosystem, and thus finally protect users. This serves to address the previously described shortcomings of certificate *revocation*. Yet, in the CA/B, no majority for this could be reached among CAs. However, Apple announced to shrink the maximum lifetime for trusted certificates to 398 days with other browser vendors following in 2020 [6, 78, 246].

Evolution of TLS ecosystem. All in all, there is a range of measures introduced in reaction to CA failures (cf. Section 2.4.2.2). Most remarkably, the evolution of countermeasures for addressing the widespread problems have radically shifted. While e.g. HPKP can provide effective protection illegitimate certificates, configuration and maintenance can be challenging and error-prone. Even more, it also introduces new risks. As it only enhances security when implemented by server operators due to low adoption, its security impact remained low. In contrast, Certificate Transparency features a broader approach. Instead of solely relying on servers to protect against attacks, CA emphasizes the role of CAs and strengthens accountability. The append-only property and distributed model of CT provides incentives for precaution and early detection in regards to malicious activity establishes an ecosystem-wide effort to enhance internet security. Likewise, the reduction of certificate lifetime seconds this by further reducing the attack surface for MitMAs.

2.5 TLS on Android

Android's network stack features a trust model comparable to browsers. Same as modern websites Android apps handle more and more privacy-sensitive information, network security is a crucial topic.

Default HTTP and HTTPS Support. Developers can use a variety of protocols for network communication in their apps. This includes out of the box support for HTTP and HTTPS. Likewise, in Android apps, this imposes the risk for MitMAs as an important attack vector especially due to internet access via (open) WiFi access points (cf. Section 2.4.2). A large portion of this risk can already be mitigated by using HTTPS. Android's HTTPS support bases on out of the box support for TLS in a similar manner as in browser environment (cf. Section 2.4.1). The Android framework implements a secure default certificate validation logic that resembles same behaviour as in browser software (cf. Section 2.4). For this, same as in browsers, Android follows the CA-PKI approach and features an immense set of root CAs that are trusted by default. Developers can take advantage of common Java APIs [146] to make HTTPS requests (cf. Listing 2.1).

Listing 2.1: Sample Code for making an HTTPS request

```
1 URL url = new URL("https://example.com");
2 HttpsURLConnection urlConnection =
3     (HttpsURLConnection)url.openConnection();
4 InputStream in = urlConnection.getInputStream();
```

Custom TLS. From the adoption of the CA-PKI approach stems the flexibility to connect to arbitrary hosts without further action needed for most domains. At the same time, this exposes Android apps and their users to similar insecurities and limitations as browsers regarding network security even when using HTTPS via TLS (cf. Section 2.4.2.2). However, the Android framework also allows developers to write custom TLS code to override the default certificate validation logic and implement their own. This way, developers can e.g. restrict trust in CAs to protect against malicious CAs (cf. Section 2.4.2.2) by utilizing pinning (cf. Section 2.4.3.1) similar to HPKP or by completely quitting trust in the PKI and using self-signed certificates or relying on private CAs not trusted by default. In contrast to HPKP and the situation in browsers, here in non-browser software like apps, developers can embed certificate pins in their app for *pinning* to check against. To achieve this, developers have to provide custom *TrustManager* and *HostnameVerifier* implementations. However, the implementation of custom certificate validation requires advanced knowledge to be conducted securely. As app developers are often no security experts, there is a substantial risk for flaws in custom implementations. Prior research has uncovered a high prevalence for insecure custom TLS [114, 126]. Listings B.1 and B.2 feature such insecure implementations. The effect of these implementations is that certificate validation is completely deactivated

Listing 2.2: Insecure TrustManager - Accepts all certificates if they are not expired

```
1 @Override
2 public void checkServerTrusted(X509Certificate[] chain, String authType
    ) throws CertificateException {
3     for (X509Certificate certificate: chain) {
4         certificate.checkValidity();
5     }
6 }
```

and any certificate is accepted for any hostname. Another dangerous example is shown in Listing 2.2. Here, for every certificate *checkValidity* is called. However, this only checks that a certificate has not yet expired. In addition, it is not checked that the presented list of certificates forms a proper certificate path [64]. While in these cases, the transmission remains encrypted, however, mounting MitMAs becomes significantly more feasible for attackers. Therefore, untrained developer will potential not find a problem, even when monitoring network traffic [115].

Summary and Outlook. Consequently, the Android framework provides developer with a large tool set to use network connectivity inside their apps. Likewise, in Android, app developers and the Android platform have similar challenges to tackle as in the browser landscape with respect to MitMAs (cf. Section 2.4.2). The Android framework provides developers with a set of APIs related to network operations and security as well as it offers them great freedom to change the process of certificate validation. This gives developers the autonomy for introducing measures for enhanced protection against MitMAs. Likewise, however, this imposes risks of flawed implementations as well as it puts developers in charge of maintenance. In this work we explore the ongoing evolution of network security in apps for the Android platform as well as we cover the impact and capabilities of developers, third parties and the Android platform itself in regards to enhancing network security in the Android ecosystem.

3
Related Work

The open-source nature of the Android operating system has led to a vibrant body of research. In this chapter, we will cover related work relevant to topics in this dissertation. Therefore, we give a comprehensive overview of research on Android and TLS (cf. 3.1) as well as the broad topic of code reuse in Android apps (cf. 3.2) to put research covered in this dissertation into perspective. In addition, we will cover further research in regard to TLS (cf. 3.3), app store analysis (cf. 3.4) and security in Android apps in a broader sense (cf. 3.5).

3.1 TLS on Android

In this section, we discuss related work regarding measurement studies of insecure TLS certification validation code in Android apps. First, we will focus on analysis approaches pertaining to insecure TLS certificate validation including our work in Chapter 5 and 6. Later on, we will also focus on alternative approaches including our proposal that we present in Chapter 4.

3.1.1 Insecure TLS certificate validation

In 2012 Fahl et al. [114] analyzed 13,500 popular, free Android apps and found 8% to be susceptible to MitMAs because of insecure TLS certificate validation code. In follow-up work in 2013, Fahl et al. [115] 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. [126] 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. [263] 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 [229] 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, Zuo et al. [313] presented a dynamic analysis effort for testing network security in hybrid Android apps. Their system utilizes static analysis to detect potential erroneous TLS error handling in WebViews. In an evaluation they analzed 13,820 apps of which they identified 625 to be vulnerable for MitMAs. In 2015 He et al. [160] presented SSLINT, a tool to detect incorrect use of TLS APIs. They found 27 previously unknown TLS-related vulnerabilities in Ubuntu applications. In 2017 Chothia et al. [85] reviewed 15 major UK mobile banking applications for Android and iOS and found that eight used pinning. However, two of them included incorrect implementations for it. Also in 2017, Fischer et al. [122] 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 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. [244] conducted a dynamic network traffic analysis with data for 1,364,420 TLS handshakes from 7,258 Android apps using 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 2017, Bojjagani and Sastry [69] present a threat model and also propose a security testing framework for penetration testing and identification of MitMA vulnerabilities for mobile banking apps (MBAs) for Android and iOS. In addition, they evaluate eight MBAs; five for Android and three for iOS.

Presented in 2018, Oltrogge et al. [P2] analyzed 13 online application generators for Android, of which six failed to implement TLS certificate validation code correctly. Chapter 5 will be based on this work and will discuss our findings. In 2018, Alghamdi et al. [15] present IoTVerif, an effort to verify TLS implementations of MQTT client apps. In 15 client apps, they find 5 to be vulnerable for MitMAs. In 2019 Kafle et al. [181] 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. [243] 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. In 2019, Wang et al. [295] present DCDroid, a combination of static and dynamic analysis for detecting vulnerable TLS implementations in Android apps. Using DCDroid, they tested 2,213 apps. 457 (20.65%) contained vulnerable TLS code. For more than half (248 apps) they could verify to be truly vulnerable using dynamic analysis. In addition, they conduct a version evolution analysis for these from which they derive that TLS related insecurities may be caused by apps becoming increasingly complex and the use of vulnerable thirdparty libraries (TPLs). This is in line with our analysis and findings in Chapter 6. In addition, we also conduct version evolution analyses to assess the adoption of NSC over time. Recently, in 2020, Weir et al. [299] 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 [114], CogniCrypt [187] and FlowDroid [52]. 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 [239] conducted work concurrent to ours [P3], in 2021, that we present in Chapter 6. They 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, in our work in Chapter 6 [P3], 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. Chapter 6 will be based on this work.

3.1.2 Alternative approaches to custom TLS

Previous works suggested alternative approaches to custom TLS to overcome the disastrous state of network security in Android apps especially regarding custom TLS implementations. As an example, in 2013, Fahl et al. [115] proposed a configuration approach to custom TLS behaviour on Android, removing the need for developers to write any code. Their approach securely covers all certificate-related use-cases found in applications or requested by developers. It includes the distinction between developer devices and end-user devices and allows for alternative TLS certificate validation strategies without developers having to change their applications. In 2013, Conti et al. [94] proposed MITHYS, a proxy-based approach to protect users from applications vulnerable to MitMAs, which is also available for Android as MITHYSApp. In 2014, Tendulkar and Enck [274] suggested a similar scheme with two main objectives to reduce vulnerabilities in custom TLS solutions: They recommend linking the debug state to TLS certificate validation as well as stating certificate or CA pinning in the application manifest. When evaluating 13,000 applications from Google Play, they found 1,889 (14.50%) would benefit from their proposals. With CERTSHIM, Bates et al. [65] presented a dynamically linked shared library that utilizes binary instrumentation to layer additional security on top of existing TLS libraries like OpenSSL or GNUTLS. It adds hooks for TLS handshakes and verification functions to force secure defaults, to protect against insecure implementations and to enable trust enhancements such as Convergence, [207] DANE [166] or pinning. CERTSHIM was successfully tested with 94% of the most popular Ubuntu packages without changes to existing applications. In an effort to provide simpler but more secure APIs, in 2015, Amour and Petullo [19] introduced libtlssep. This library aims to prevent common mistakes by developers who write TLS-related code with existing TLS implementations while still being easy to integrate. Oltrogge et al. [P1] analyzed the applicability of pinning in Android apps to be less than two percent. Likewise they only found a small fraction of apps that implement pinning in app code. To understand the knowledge gap of developers regarding pinning, they conducted a developer study. Based on that, they propose a tool to support developers in the decision process for and implementation of pinning. Chapter 4 will be based on this work as part of this dissertation. In 2016, Buhov et al. [74] strive to tackle the urgent state of erroneous TLS implementations in Android apps. In contrast to previous strategies for mitigation of vulnerable TLS code e.g.

by Tendulkar and Enck [274], Fahl et al. [115] or Oltrogge et al. [P1], they propose a system-based instead of developer-driven approach for certificate pinning. This is strongly related to our work presented in Chapter 4 [P1]. In addition, as we identify the challenge to maintain certificate pins due to slow propagation of app updates as a crucial limiting factor for the adoption of certificate pinning (cf. 4.6.3), they propose a system-based dynamic instrumentation for maintaining certification data for pinning and fixing of errors without any dependency on developer action. To accomplish this, they realize pinning on the basis of TOFU and handling encountering new pins or certificates respectively in a browser-like fashion in a dialog for users to confirm. While this shifts responsibility away from developers, it puts an even greater burden on end users again compared to the already well-studied and researched complexity regarding TLS warning dialogues in browsers (e.g. [14, 272, 120, 118]). In 2016, Wei et al. [298] present emphaSSL, a tool to support developers in hardening network security in their apps. Based on PMD[1] [237], emphaSSL employs a rule set for verifying network security implementations in code to give developers instant feedback on misuse of TLS APIs during development to allow for immediate fixing and protect against potential risks for MitMAs. They evaluate 75 open source apps using their effort. In 30 they identify potential mistakes pertaining to network security of which they deem nine as seriously and rendering apps vulnerable to MitMAs. This is roughly comparable to Android Studio's limited LINTING features regarding TLS (cf. [281, 169]) for which we argue the need for improvement for better support in Chapter 6. In a more recent approach in 2017, O'Neill et al. [226] introduced TrustBase, a centralized approach to move TLS certificate validation to an OS service that intercepts all connections and enforces policies. They use a plugin infrastructure to provide features such as pinning. They demonstrate their proposal as a loadable Linux kernel module and describe prototypes for both Android and Windows. In 2017, Nguyen et al. [224] presented FixDroid. FixDroid addresses shortcomings in Android's LINT utilities (e.g. quick fixes soliciting the use of HTTPS).

3.2 Code Reuse in Android apps

Chapters 4 and 5 will deal with analyses of legitimate reuse of app components in Android apps. Earlier efforts featuring TPL detection or making use of these rely on whitelisting approaches to detect libraries based on their package name.

3.2.1 Library Detection

In 2012, Grace et al. [155] presented AdRisk, a system to identify potential risks in ad libraries. They analzed 100,000 apps and found 52,067 (52.07%) to have at least one embedded ad library. They searched 100 ad libraries based on package names. In addition, they used AdRisk to assess the risks introduced by these libraries systematically.

[1] Programming Mistake Detector

This includes access *dangerous* APIs for collection of personal for advertising purposes ranging from location data to even more sensitive data like phone number, browser bookmarks and call log information that is sent to the ad providers' server. For five ad libraries, they identified the dangerous practise of dynamically fetching and executing code from remote. Stevens et al. [270] seconded this and analyzed privacy risks imposed by ad libraries. In 2012, Pearce et al. [233] proposed AdDroid, an extension for the Android System to separate application and ad library code to address over-privileged ad libraries and to expose separate APIs related to advertising for ad libraries to interact with. For evaluation, they split application code into application code and ad library code by their namespaces. In 2013, Book et al. [70] measured the use of privacy risky permissions in ad libraries over time. They analzed 116,000 apps and considered 68 different ad libraries for detection by package name. They extracted the corresponding library code from apps for which they conduct hashing in order to derive unique identifiers for different versions of ad library. Based on this, they showed an increasing use privacy sensitive permissions. Aside from criticizing this evolution, they stipulated a separation of app code and library code to both limit the permissions as well as to relieve from the need of having developers embed library code and rely on them to update for newer versions. Book and Wallach [71] base on [70] and analyzed privacy leakage of 20 most popular ad libraries in Android apps. In 2013, Gibler et al. [129] presented AdRob, an analysis on the impact of app plagiarism on ad revenue. For detection of ad libraries, they searched for *client IDs* of ad libraries and conducted network traffic analysis on these to assess impact of app plagiarism. Regarding assessing original ownership of apps they analyzed signing certificates of apps. Their analysis accounts for 265,359 free applications from 17 Android markets. As the first efforts for automatic detection of ad libraries Narayanan et al.'s [218] AdDetect in 2014 and Liu et al.'s [201] PEDAL in 2015 used machine learning techniques. Most of these efforts are tightly related to ad libraries. Their motivation spans from privacy risks and vulnerabilities introduced by ad libraries [155, 270, 70, 71] to detection of app clone with respect to ad revenue stealing or cloned apps containing malware [129]. In contrast to detecting TPL-related code for specific libraries as most efforts pursue in regards to ad libraries, our goal in Chapter 4 is to distinguish custom app code from arbitrary TPLs. In that, we adapt the approach of white-listing libraries for detection of arbitrary TPLs by checking for exclusiveness of package names (cf. 4.4.3.3). Since then, there have been several efforts and advancements in regards to detecting TPL in Android apps. In 2015, Wang et al. [292] proposed WuKong, a two-phased approach for detection of repackaged apps. In 2016, Li et al. [197] analyzed 1.5 million apps from Google Play and curated two white-lists for 1,113 common libraries and 240 common libraries identified as ad libraries. One important limitation of previous efforts is that these are not resilient to even basic obfuscation techniques such as renaming of packages, classes or methods. In 2014, Linares-Vásquez et al. [200] already hint at an impact of obfuscation on the accuracy of detection of code reuse. Even more, in 2018, Wermke et al. [302] presented an analysis on prevalence and advancement of obfuscation techniques in apps. Later research efforts tackle these limitation. In 2016, Ma et al. [204] presented LibRadar, a tool for detecting TPL based on stable API features derived from one million free apps relieving from the reliance on limited whitelists of known TPLs. In 2016 Soh et

al. [261] proposed LibSift. LibSift's approach for detecting TPLs is bases on package dependencies. In addition, they compared their approach with LibRadar [204] and Li et al. [197] in an evaluation of 300 apps and find LibSift to detect even less prominent TPLs not detected by either of the other two approaches. In 2016, Backes et al. [57] presented LibScout. LibScout employs a merkle tree-based approach for detection of arbitrary TPLs with focus on detecting malicious or otherwise outdated TPL versions and their prevalence in apps which can harm app and user security if not updated by developers. One particular worrisome example in regards to TLS is a vulnerable version of OkHttp [99] where use of pinning renders apps vulnerable to MitMAs. Apart from this, in Chapter 5, we will employ LibScout to explore the extent to which OAGs work with boilerplate code. The line of research on TPL detection is extended by further efforts. In 2017, Li et al. [199] presented LibD. In 2018, Zhang et al. [310] proposed LibPecker. Han et al. [167] proposed a name and feature based approach to detect libraries in Android apps. Wang et al. [294] presented ORLIS. In 2019, Zhang et al. [309] presentec LibID. In 2021, Almanee et al. [18] presented LibRARIAN. In 2020, Zhan et al. [308] conducted a comparison taking several of these efforts into account. From their results, they derive a further need for enhanced techniques such as to address Inversion of Control (IOC) patterns like dependency injection (DI), dynamically loaded TPLs or use of alternative programming languages like *Kotlin* that feature different semantic than Java.

3.2.2 App Generators

After giving an overview pertaining to research on TPL detection, we want to focus on related work concerning our coverage of application generators in Chapter 5. In 2016, Gonzalez et al. [131] presented CodeCheck, a system to analyse code reuse and uniqueness. They analyzed 60,000 apps. They extract opcodes from DEX codes that are then hashed with Message-Digest Algorithm 5 (MD5) to be correlated and analyzed by different categories. These categories also include TPLs and sorts of app generators. Also, in 2017, Glanz et al. [130] presented LibDetect and CodeMatch as a two step effort on detecting repackaged apps by first removing library code from obfuscated apps. In this regard, they identify app generators as a source of legitimately shared code which therefore poses a challenge for detecting repackaged apps. While these efforts already recognize app generators, prior work did not thoroughly investigate this topic. Though, in 2016, Li et al. [198] further motivated research on this as necessary. They are first to deal with and foresee an emerging trend for app generators by exploring practises of Software product line (SPL) in software engineering of apps. Regarding this, they proposed case studies for extractive SPL adoption to explore reuse practices in app families. This is strongly related to our analysis in Chapter 5. The characteristics they observed resemble those identified by us regarding the detection of app generators. However, in contrast to the approach of mining app families by observations on e.g. similar package names and signing certificates, our effort bases on a comprehensive market analysis (5.3.1). Moreover, back in 2013, Fahl et al. [115] – in addition to their proposals (cf. 3.1.2) – also already mentioned application generator frameworks

Apache Cordova [2] [30] and Appcelerator's Titanium [35] software development kit (SDK). They uncover dangerous configuration settings that deactivate proper TLS certificate validation rendering apps vulnerable to MitMAs even without developers writing any TLS-related code on their own. This already serves to illustrate the massive impact that application generators can have on the security of the ecosystem. In Chapter 5, we will introduce these frameworks as Standalone Frameworks (SAFs). There, we will further elaborate on application generators and their security impact. We will focus on OAGs which take full control over the app creation process and thus have complete responsible for guaranteeing security. Acar et al.'s poster [9] presents preliminary results of our work and already hints at dangerous security practises of certain OAGs, e.g. regarding security of custom TLS code and Inter-component communication (ICC).

3.3 Transport Layer Security

Giving a complete and comprehensive overview on TLS, CAs and PKI trust models in more detail than already introduced in Chapter 2 would be beyond the scope of this thesis. For a broader overview, we refer to Clark and van Oorschot's Systemization of Knowledge (SoK) on SSL and HTTPS. [89] In 2013, Durumeric et al. [103] present results of 110 scans of IPv4 address space. They use ZMap [104], their internet-wide scanning approach for surveying the entire IPv4 address space in less than 45 minutes. From the collected data, they can derive TLS-related information on hosts. This includes protocol adoption of HTTP or HTTPS and TLS certification data which also allows e.g. detection of misissued (CA) certificates. In 2014, Perl et al. [234] investigate the root problem of the weakest-link property of the PKI. In an analysis of over 400 pre-installed CAs on Windows, Linux, MacOS, Firefox, iOS and Android they check against an extensive database of 47 million certificates collected from HTTPS servers, they find only 66% of the CAs to have signed certificates of HTTPS servers. Based on these, they discuss risks and benefits of removing the other CAs from trust stores especially as a measure to increasing security of CA-PKI. In 2014, Vallina-Rodriguez et al. [289] explore the CA trust store on Android and elaborate implications on users. They find a variance in the set of trusted CAs on users' devices also depending on manufacturers, mobile carriers or even apps. In addition, as Android does not differentiate between user and system trust store, CAs installed by users are trusted same as system CAs and lead to a bigger attack surface for MitMAs. For mitigation, they recommend an auditioning for inclusion of CAs in devices' trust store and question if users awareness regarding making trust decisions on their own. As part of further development of Android, one of the changes in Android 7 is such a differentiation. Since then, app developers have to trust user-installed CAs explicitly. In Chapter 6 we will further investigate this. In 2017, Felt et al. [119] measure the adoption of HTTPS from a user and developer perspective. As to the adoption of HTTPS, they find a substantial growth over the last years. Whereas, they found HTTPS adoption on mobile devices to lag behind. While, as opposed to our work in regarding network security in apps, their measurement

[2] Formerly PhoneGap

deals with adoption of HTTPS in mobile browsers, we, likewise, find a need for steeper increase. While topics related to cryptography of TLS are out of scope of this thesis, we refer to Kotzias et al.'s [183] large-scale study on the evolution of TLS deployment over the last six years with focus on high-profile cryptographic attacks.

3.4 App Market Analysis

There is a variety of efforts on crawling app stores for Android apps. Early works considered a variety of app store resources for app mining. In our work, however, we will exclusively focus on Google's ecosystem for Android (cf. Chapter 2) and present our own crawling effort (cf. 2.3). Our approach is however vastly comparable to efforts by Viennot et al. [291] in 2014, Carbunar et al. [76] in 2015 or Allix et al. [17] in 2016 concerning collection of metadata and downloading APKs from Google Play [148]. This falls in the same time period as the development of our approach.

3.5 Overall security in Android Apps

Apart from the in-depth coverage of related work concerning topics in this thesis, there is a large and growing body of research pertaining to Android. The focus of our work lies on TLS development in Android apps. However, our work in Chapter 5 is strongly related to manifold research efforts. Therefore, to give a broad overview regarding Chapter 5, we refer to seminal work on *Overprivileged Apps* (e.g. [116, 53]), *ICC Security* (e.g. [109, 311, 202, 153]), *Cryptographic API Misuse* ([106]), *TLS APIs* (cf. 3.1) *WebView Security* (e.g. [127, 83, 203, 293, 277]) and *Code Injection* (e.g. [238]). The corresponding sections in Chapter 5 will relate in more detail to these topics. For more insights, we refer to Acar et al.'s [7] SoK on Android in the context of security and privacy research.

4
Developers

Exploring applicability of pinning

4.1 Introduction and Contributions

As already covered in Chapter 3, previous research uncovered security issues with TLS in mobile apps [114, 115, 126, 65, 263] that highlight that developers have problems with implementing correct certificate validation while users are challenged by TLS interstitials. Furthermore, the default TLS implementation on Android receives criticism [289, 234]: Adopted from web-browsers, default TLS certificate validation relies on a huge number of root CAs pre-installed on all Android devices [289]. Hence, all Android apps suffer from the same issues as web-browsers: A single malicious CA is able to conduct MitMAs against all apps trusting the respective certificate. Chapter 2 already introduced to this. To make things even worse, Fahl et al. [115] uncovered that in 97% of all cases where developers implemented their own certificate validation strategy, they turned off validation entirely and left their apps vulnerable to MitMAs with arbitrary certificates, i.e. every active network attacker was able to attack successfully.

Pinning is often recommended as a general countermeasure to tackle the weakest link in the CA-based infrastructure [31, 209, 231, 115]. We use the term *pinning* in this work to include both pinning the complete X.509 certificate or only the certificate's public key. Instead of trusting a large set of root CAs that come pre-installed with the operating system, software limits the set of certificates it trusts to *pins*, which can be single leaf certificates, single root CA certificates or a set of certificates. Pinning is a straightforward mechanism and its deployment does not require changes to the current CA infrastructure. However, pinning has not found widespread adoption yet. While limiting the number of trusted certificates drastically increases security, pinning doesn't come for free: Embedding trusted certificates into an app requires app updates whenever the pins change. Hence, the decision whether pinning is applicable for a TLS connection is always a trade-off between increased security and maintenance effort that is not entirely under the control of an app developer: Whenever users do not update their app although the pins have changed, the app stops working and users might uninstall the broken app. Therefore, to pin or not to pin is a critical decision for app developers, which requires in-depth understanding of the mechanisms behind pinning and its implications on their apps. This is where our work comes in: To the best of our knowledge, we are the first to explore the applicability of pinning as an appropriate alternative certificate validation strategy for non-browser software. In this work, we focus on individual app developers' capabilities for enhancements to network security and make the following contributions:

Status Quo. We evaluate the status quo and analyze 639,283 Android apps to find that only 45 apps implement pinning.

Formalization. Instead of intuitively recommending pinning, we formalize criteria that must be considered when the decision is made whether to pin or not to pin.

Implementation. We apply static code analysis and program slicing to automatically assess those criteria and obtain an overview of the situation for single apps.

Evaluation. We evaluate our criteria against a set of 639,283 Android apps for an overview of the applicability of pinning in the Android universe. We find that 223,655 apps establish TLS-secured connections to remote origins; 11,481 (5.13%) of these 223,655 apps are eligible candidates to implement pinning for one or more of their TLS connections.

App Updates. Since new certificate pins need to be updated on the users' devices, the update speed is crucial. Therefore, we instrument telemetry data from a popular anti-virus software provider. We evaluate the update behaviour of 871,911 unique users from January 2014 to December 2014 and find that only 50% of the users update to a new app version within the first week after release.

Developer View. Although pinning is only applicable in relatively few cases, the nominal-actual comparison leaves room for improvement. We therefore collected feedback from 45 developers of apps for which we would recommend pinning. We identified the developers' major issues with pinning and used their feedback as the foundation to build an easy-to-use web application that assists developers with securing their apps' TLS connections. We offer help on the decision whether to pin or not to pin and support the implementation of pinning with concrete suggestions and code examples.

Take-aways. We formulate lessons learned during our evaluation to share them with the research community.

4.2 Background

The Android platform provides built-in functionalities for handling TLS and certificate validation based on the PKI without further configuration. It comes pre-loaded with an extensive truststore featuring 140+ root CAs [234]. Additionally, the Android framework enables developers to provide custom implementations for handling certificate validation. Chapter 2 already elaborated on these fields of problems and challenges in the TLS ecosystem (cf. 2.4), in addition to introducing to the situation on Android (cf. 2.5). There are several reasons for developers to use custom implementations:

- Application developers might want to use a self-signed certificate either for testing, effort- or economical reasons;
- When a root CA is not in the system-wide list of trusted root CAs, a company might have an internal CA that issues certificates for use in intranet applications;
- Security can be enhanced by restricting reliance on the PKI to mitigate the exposure to weaknesses in the PKI (cf. 2.4.2.2)
- and custom implementations are required to implement leaf or CA pinning.

4.3 An Exit Strategy

Previous work shows that TLS is complex and error-prone (cf. 3.1). Studies imply that the implementation of client code for correct certificate validation is hard for software developers. The general trust model received a lot of criticism over the last years and alternative solutions have not found widespread adoption.

As a general solution for both problems, pinning is often advocated as a secure and reliable alternative to the default trust model for non-browser software [231, 209, 31, 115]. Pinning can serve two purposes. First, it can mitigate the risk of MitMAs as it strengthens the validation process. Secondly, pinning allows to overcome limitations of the current CA infrastructure, e.g. by allowing self-signed certificates.

We challenge the recommendation to use pinning for non-browser software and conduct a deep analysis on the root causes that hinder its widespread adoption in non-browser software.

4.3.1 Pinning in Android Apps

Before evaluating the status quo, we give a brief overview of how TLS pinning can be implemented in Android apps. In general, Android apps can establish low-level TLS connections via an `SSLSocket` or an `HTTPSUrlConnection` object. Using pinning in these cases requires the developer to implement a custom version of Android's `TrustManager` interface with an appropriate `checkServerTrusted` method. This method has to check whether the certificate sent by the remote server matches one of the given pins. Another option is to use a custom `KeyStore` in which developers can store their own certificates. Many Android applications do not make use of such lower level APIs but use a `WebView` to display HTML directly. Sadly Android does not provide an API to implement pinning for `WebViews` [296]. As a workaround, developers can download all HTML/JavaScript via the low-level `HTTPSUrlConnection`, store it locally and only use the `WebView` for rendering. However, this is a clear shortcoming of Android's API and makes the implementation of pinning unnecessarily hard.

4.3.2 Status Quo

For a better understanding of the current adoption of TLS pinning in Android apps, we evaluated custom certificate verification strategies for 639,283 Android apps. Therefore we used `MalloDroid` [114] and extended the classification feature of custom verification implementations. We focused on both customized implementations of the `TrustManager` interface and the usage of a custom `KeyStore` for TLS certificates.

Whenever we found that a `KeyStore` object was created, we conducted a reachability

analysis [202] for this object. For objects that were reachable from an app's entry point, we assume that this app uses pinning. Next, we extracted the keystore file that was loaded to check whether leaf or CA certificates were pinned. We found 21 apps that implement pinning using the keystore method. 13 of these apps pin a leaf certificate, while 8 of them pin a custom root CA certificate.

Whenever we found a `TrustManager` implementation, we checked whether the `chain` parameter of the `checkServerTrusted` method was accessed by the implementation. Implementations that do not use this parameter do not verify the remote origin's certificate chain and hence were removed from further analyses. In a second step we conducted a reachability analysis for implemented `TrustManager` objects and removed all implementations that were not reachable from an app's entry point. We found custom `TrustManager` implementations in 42,902 apps and could remove 42,042 apps from the list since they either implemented bypassing `TrustManagers` or were not reachable. The remaining 858 apps implemented 189 different `TrustManagers`. We compared these implementations with the list provided by Fahl et al. [115] and could filter out 124 implementations that basically add logging for certificate validation. We manually reviewed the remaining 65 implementations and found that 13 implemented pinning. Overall, these implementations were used by 24 apps.

Altogether, of the 639,283 apps in our data-set, 45 implement pinning. These numbers confirm findings already reported by Fahl et al. [115].

4.4 Classification Strategy

The decision whether or not a TLS connection should be secured by pinning depends on multiple factors and is not trivial in many cases.

Furthermore, whenever we cannot reliably identify the origin string for a TLS connection endpoint, we cannot assess whether pinning would be a reasonable validation strategy. Therefore, we report our results for two different scenarios:

Conservative. We report numbers for a conservative scenario. Whenever we cannot identify the origin string for a TLS connection endpoint, we assume that pinning is unfeasible. This covers most of our results.

Optimistic. For some of the cases where we could not successfully identify origin strings, pinning could be applied under certain circumstances. These cases are treated differently for the optimistic scenario as detailed in Section 4.6.

For our classification, we consider the following properties as high level indicators:

Prior Knowledge of the Target Origin. Prior knowledge about the target origin is vital: Pinning is only feasible if the developer of an app is able to hard-code target

origins into their app. This includes adding target origins at compile time as well as via configuration files before or at run-time. Whenever target origins depend on user- or external app input – e.g. via an `Intent` – at run-time, we consider pinning as not feasible, since web-browsers do not automatically pin certificates for websites for the same reasons.[1]. Automatically pinning previously unknown origins would increase the danger of failed TLS handshakes due to substituted certificates and would decrease acceptance of pinning for both developers and app users.

In the *conservative* scenario, we recommend the default validation strategy for all connections where the origin depends on external input. However, some of these connections can be pinned in the *optimistic* scenario (cf. Section 4.6).

Ownership of Relevant API Calls. App development consists of writing one's own code and embedding external libraries. All source code that was written by the developer or the developer's company is considered owned by the developer. API calls required for certificate validation during the establishment of TLS-secured network connections are *relevant* for this. Relevant API calls might be a `HTTPSUrlConnection` or an `SSLSocket`. Whenever ownership of relevant API calls is given, pinning might be feasible. Library developers do not own their code when it is included in other apps. Therefore, we do not recommend pinning for library code. Library developers cannot control when app developers update their libraries, while app developers can hardly influence whether library developers keep their certificate pins up to date.

For both the *conservative* and the *optimistic* scenario pinning is not recommended in case API ownership is not given.

TLS Certificate Configuration Responsibility. Being responsible for the TLS certificate configuration as well as being the owner of relevant API calls eases the coordinated deployment of pinning in apps. In case developers or their companies have control over the TLS certificate configuration of origins used in apps, both the certificate pins in apps and the corresponding server configurations can be coordinated. In these cases, pinning is feasible. Whenever apps communicate with public origins, such as public API interfaces or websites, pinning cannot be recommended. Certificate configurations can change frequently and the responsible administrators only rarely announce them in advance. Unplanned certificate changes can lead to failing TLS handshakes and are therefore unacceptable.

For both the *conservative* and the *optimistic* scenario pinning is not recommended in case the developer is not responsible for TLS certificate configuration.

[1]Administrators can configure HTTP Public Key Pinning to pin TLS certificates in modern web-browsers. However, this involves heavy manual configuration work on the server side and does not happen automatically (cf. Section 2.4.3.3).

4.4.1 Possible Recommendations

The above criteria build the foundation for the decision whether pinning is applicable for a TLS connection in a given app. The classification algorithm recommends one of the following strategies:

Leaf Pinning. Leaf pinning limits the number of trusted certificates to the server's leaf certificate/public key (cf. Section 2.4.3.1).

CA Pinning. CA pinning effectively limits the number of trusted CAs (cf. Section 2.4.3.1). We treat conventional pinning (cf. Section 2.4.3.1) and TOFU (cf. Section 2.4.3.2) equally, since both provide a similar level of security (cf. Section 2.4.3.1) and require the same maintenance overhead from an app developer's point of view.

Default. Whenever none of the above criteria applies, pinning is not a recommended strategy and should not be implemented. This also accounts for cases where external input – e.g. user input in an address bar or `Intent` input – influences a TLS connection.

4.4.2 Classification Details

Algorithm 4.1 illustrates the classification process we apply to decide whether pinning is an advisable verification strategy. In the initial state, the default strategy is assumed to be the right choice for a TLS connection, since we do not know yet if pinning is recommendable. First, we check whether the TLS connection is established within a third party library. In this case classification ends with the recommendation to us

Second, we check whether the remote origin depends on user input, other external sources such as `Intents`, or may be configured via a configuration file. In these cases we recommend the default strategy, since control over the remote origin is not guaranteed.

Third, we check whether the TLS connection's remote endpoint is a popular origin; in this case, classification ends with recommending the default strategy, since the TLS configuration of the remote origin is probably not under the developer's control. If we assume that a remote origin is probably under control of the developer, as no other app accesses it, the classification continues: We check whether the origin's certificate was issued by a valid CA or is self-signed. For self-signed certificates and certificates that were issued by a valid CA, we recommend leaf certificate pinning. For certificates issued by an untrusted CA, we recommend CA pinning, since the custom CA is probably under control of the developer.

Algorithm 4.1: The Classification Process.

for $r \in results$ **do**
  $dependencies \leftarrow dependencies(r)$;
  set strategy as default;
  if $dependencies \neq \varnothing$ **then**
    if $\exists d \in dependencies | d \in \{exposed\,intent, UI - Component\}$ **then**
      continue;
    else if $\exists d \in dependencies | d \in \{unexposed\,intent, variable, configuration\}$ **then**
      continue;
  if not *callInLibrary(r)* **and not** *isPublicHost(r)* **then**
    $host \leftarrow host(r)$;
    $schema \leftarrow schema(r)$;
    $cert \leftarrow cert(r)$;
    $dependencies \leftarrow dependencies(r)$;
    if $isUnderControl(host)$ **then**
      if $signedByUntrustedCA(cert)$ **then**
        mark for CA pinning;
      else if $validCertChain(cert)$ **or** $isSelfSigned(cert)$ **then**
        mark for leaf pinning;

4.4.3 Challenges

For our strategy classification, we apply static code analysis on a large set of Android apps. To work as efficiently as possible, we identified multiple challenges:

4.4.3.1 Relevant API Calls

First, we identify relevant API calls, which means taking remote origins as parameters and establishing a TLS-secured connection between the app and the origin. The official Android API documentation identifies relevant API calls in the packages presented in Table 4.1.

These API calls are the most low-level calls in the API and they implicitly include higher level APIs such as the `HTTPSUrlConnection`.

Package name
org.apache.http.client.methods.*
org.apache.http.HttpHost
android.webkit.WebView.loadUrl
android.webkit.WebView.loadDataWithBaseURL
android.webkit.WebView.loadData
android.webkit.WebView.postUrl
android.net.http.AndroidHttpClient
java.net.Url

Table 4.1: Relevant API Calls.

4.4.3.2 Embedded Static TLS Origins

As described above, knowing remote origins in advance is crucial for pinning. At this point, we focus on extracting whether a remote origin is embedded in an app or depends on user input or is injected via an external interface such as an `Intent`. This information is supplied via parameters to relevant API calls. Although these mainly refer to `String` values, the object-oriented and Java-based nature of the Android platform introduces complexity:

- `Strings` may not be constant values but can be composed of numerous substrings. We identify concatenation of Strings, formatting of `Strings` and platform-specific APIs for building URIs as relevant. Therefore, we statically backtrack these and reproduce `String` values.
- Values can stem from variables or may be return values of method calls. Therefore, we account for intra-application method calls as well.
- Values that stem from `Resources`, `Properties` or `Preferences` hint at configuration parameters.
- Origins can be input parameters for application entrypoints. Entrypoints are parts of an application that allow either other app components, external apps or users to interact with an app. Android application entrypoints are `Activities`, `Services`, `Receivers`, `Intents` and `Bundles`. UI-Components in `Activities` hint at user input.

4.4.3.3 API Call Ownership

API call ownership is a requirement for pinning. To identify whether an app developer holds ownership of relevant API calls, we must distinguish relevant API calls that are accessed by third party libraries and relevant API calls accessed by code that was

written by the app developer (we call this *custom code*). Whenever we find relevant API calls in code that is shared between multiple apps by different authors, we assume a library that is not under control of an app's developer.

4.4.3.4 Origin Exclusivity

Whenever the TLS configuration of a remote origin is not in the range of influence of the developer, pinning is not advisable. We classify origins that are shared between multiple apps' authors and connections that access public origins as not under control of the developer.

4.5 Implementation Details

To decide whether pinning for a TLS connection is advisable and to address the above challenges (cf. Section 4.4.3), we implement our classification strategy in a multi-step process. We extend the MalloDroid tool [114, 205] and execute the following steps:

1. Disassemble a given Android application to gain access to the application's code and call graph (cf. Section 4.5.1).
2. Identify relevant API calls – i.e. API calls that implement TLS connections in apps (cf. Sections 4.5.2 and 4.4.3.1).
3. Extract information for remote origins applying program slicing (cf. Sections 4.5.3 and 4.5.3.1).
4. Determine whether API and/or origin ownership for relevant API calls is given and decide which certificate validation strategy suits best (cf. Sections 4.4.2 and 4.5.4).

4.5.1 Disassembly

In Step 1, we use androguard [22] to disassemble apps and construct call graphs for further processing.

4.5.2 Relevant API Calls

In Step 2, the call graphs were used to identify apps that make use of API calls in which origin information for establishing TLS-enabled connections is specified (cf. Section 4.4.3).

We consider API calls as *relevant* if they are used during the process of TLS connection establishment (cf. Table 4.1).

4.5.3 Program Slicing

In Step 3, we apply backwards program slicing [300] to collect method parameters which we subsequently call *slicing criteria*. We focus on slicing criteria that represent remote origins which are used as input for relevant API calls, i. e. we are backtracking parameters that are URLs or hostnames for TLS connections.

Our approach is similar to the one applied by Poeplau et al. [238], who apply a backward slicing algorithm to identify security issues with dynamic code loading in Android apps.

In contrast to backwards slicing single and fixed method parameters, our targets are network origins. A network origin `String` can be a composition of multiple substrings. We take this fact into account by applying backwards slicing for multiple parameters. After backtracking, we join these (multiple) substrings to one origin string whenever possible. To break cycles, we stop the slicer after 80 iterations, which guarantees that the algorithm terminates and also makes sure we do not lose data.

4.5.3.1 Extracting Origin Strings

Origin strings can be compositions of multiple substrings (e.g. `https://` and `www.example.com` and `:443`). Thus, reconstructing an origin string might require combining multiple sliced substrings. Therefore, after program slicing, we apply a combination of backward and forward analysis. Backwards analysis is used for backtracking constant register values while forward analysis determines calls of a `String` instance. Both, back- and forward analyses are applied multiple times successively as long as we find new substrings that are part of a final origin string.

Algorithm 4.2 outlines pseudocode for handling `StringBuilder` objects. The algorithm makes use of the following functions:

methodsOnInstance returns a list of all method invocations on an instance (e.g. calls to `toString`, `append` or the constructor for `StringBuilder` objects).

backtrack applies the actual backtracking techniques to gather all substrings for the origin string composition.

add adds one `originSubstring` to the array of `originSubstrings` that make the origin string we are looking for.

join merges all `originSubstrings` to get the final origin string that is represented by the given `StringBuilder` object.

Algorithm 4.2: StringBuilder Analysis

```
Data: stringBuilderObjects sbos
Result: new originSubstring O
begin
    for sb ∈ sbos do
        methodInvocations ← methodsOnInstance(sb);
        originSubstrings ← ∅;
        origin ← null;
        for mi ∈ methodInvocations do
            if isAppend(mi) then
                originSubstring ← backtrack(mi.regs[1]);
                if originSubstring ≠ null then
                    add(originSubstrings, originSubstring);
            else if isToString(mi) then
                break;
        join(O, originSubstrings);
```

1. Identify instructions indicating the instantiation of a `StringBuilder` object (i.e. a `new-instance` instruction referring to the `StringBuilder` constructor) and store them in `sbos`.

2. Find all method invocations for each `StringBuilder` object `sb`;

3. For calls of the constructor or the `append()` method, backtrack the register value for the `String` parameter `originSubstring` and add it to all `originSubstrings`.

4. Stop on calls of the `toString` method and join all collected `originSubstrings` to a new `originSubstring`.

Similarly, we support `String` concatenation, formatting `Strings`, `UriBuilder` instances, Android `UI-Components`, `Intents`, `Bundles`, `Properties`, `Preferences` and `Resources`. For `Intents` and `Bundles` we identify the source in order to determine whether the corresponding component is exposed externally, e.g. via a `Service`.

4.5.4 Decide on Validation Strategy

In the final Step 4, we assess whether API and/or origin ownership is given (cf. Section 4.4). For pinning candidates, X.509 certificate information is collected and a decision for or against pinning is made (cf. Section 4.4.2).

4.5.5 Limitations

The described approach is limited in multiple ways. We decided to reverse engineer a large sample of free Android apps and analyze the resulting code. This limits the analysis compared to the analysis of original source code, e.g. we lose variable names and cannot preclude obfuscation. This is however the state of the art for large scale app analyses, since reaching out to developers and asking for source code does not scale well. We apply static code analysis and program slicing to determine the best certificate validation strategy to secure a TLS connection. Our approach does not consider native code in Android apps. We cannot track potential TLS connections in code that was dynamically loaded or when obfuscation was applied.

Since we apply static code analysis techniques, we might report some false positives: Some of the TLS connections we found and identified as being reachable code might not be used during real application usage. However, this is not a specific limitation of our work, but a general limitation of static code analysis.
We might also report some false negatives: Due to the strategy to classify origins to which apps developed by different developers connect as "not under control of the developer", we might miss the scenario that one company has several distinct developers write apps for them that all access the same domain. However, there exist no criteria to distinguish these cases from the common scenario that multiple apps by different developers accessing the same domain means that the domain is not under control of the developers. Therefore, these cases are included in the group of public origins for which we do not recommend pinning.

4.6 Evaluation

We applied the classification algorithm (cf. Section 4.4) to a set of 639,283 Android applications we downloaded from Google Play in October 2014. Our data extraction showed that of these apps, 573,258 implemented network connections.

In the following, we report details of our automated large-scale analysis. We report our results on a per-connection base (see Figure 4.2) as well as on a per-app base (see Figure 4.1), where an app is counted as eligible for pinning if at least one of its connections can be pinned. We distinguish between a conservative and an optimistic strategy rating (cf. Section 4.4). Altogether we found 20,020,535 calls to network related API calls (cf. Table 4.1). For these calls we tried to identify the origin strings. We could identify 1,062,810 calls as TLS connections due to the fact that the corresponding origin string's scheme was HTTPS. 2,420,104 connections were identified as plain HTTP, while 16,537,621 connections did not have a hard-coded origin string in the app's code. Hence, for 81% of all connections, it was not directly possible to determine whether TLS was used. However, a deeper analysis based on our classification criteria (cf. Section 4.4.2) gives detailed insights into the applicability of pinning. Table 4.2 gives an overview of the results.

Overall	**Con.**	**Apps**
Hard-coded HTTPS Origin	1,062,810	229,317
Hard-coded HTTP Origin	2,420,104	414,194
Non-hard-coded Origin	16,537,621	553,399
	20,020,535	573,258

Third Party Libraries	**Con.**	**Apps**
Hard-coded HTTPS Origin	917,567	203,159
Hard-coded HTTP Origin	1,659,933	310,331
Non-hard-coded Origin	14,564,581	512,055
	17,142,081	517,790

Custom Code	**Con.**	**Apps**
Hard-coded HTTPS Origin	145,243	48,755
Hard-coded HTTP Origin	760,171	184,184
Non-hard-coded Origin	1,973,040	246,636
	2,878,454	299,863

Table 4.2: Distribution of Network API Calls.

4.6.1 Library Code

The majority of network connections we identified were made in third party library code, i. e. users include third party libraries to make use of external functionality. Such connections can include both plain and TLS-protected connections. As described in Section 4.4.2, we recommend not to use pinning for those TLS connections, as API ownership is not given. Of the 20,021,137 TLS connections we could identify, we found that 17,142,081 (85.6%) connections are embedded in third party libraries.

Table 4.3 gives an overview of the top 10 third party libraries we found in our data-set.

Most of the identified libraries belong to ad networks (e.g. `com.google.ads.*`), crash reporting tools or app building kits (e.g. `org.apache.cordova.*`) that establish network connections without any interaction with an app's custom code.

We found the `AndroidPinning` library [209] to be the only library that supports pinning as a security feature out of the box. However, in our data-set we found only 14 apps that made use of this library (cf. Section 4.3.2).

Library	Connections
com.google.ads.*	2,535,020
org.apache.cordova.*	1,145,108
com.qbiki.*	977,298
com.millennialmedia.*	730,408
com.facebook.*	551,373
com.Tobit.*	488,143
com.inmobi.*	352,855
com.flurry.*	340,929
com.startapp.*	276,988
com.adsdk.*	234,130

Table 4.3: Top 10 Third Party Libraries.

4.6.2 Custom Code

Next, we identified network connections that were established in code that was actually written by apps' developers, i.e. network-related API objects were not instantiated within third party libraries.

We found 2,878,454 connections in custom code of which we identified 145,243 as TLS connections. 48,755 apps implemented hard-coded TLS origins as parts of their custom code. Based on the type of the deployed certificate and depending on whether the origin is shared, we evaluated which of these TLS connections are candidates for pinning.

For 1,973,040 of the connections that were implemented as part of apps' custom code, we could not identify hard-coded origins in apps. Those connections could be either HTTP or HTTPS and depend on further input available only at run-time, e.g. user input or Intents. For these connections, we consider a conservative as well as an optimistic scenario. Based on different assumptions, these scenarios allow us to estimate the applicability of pinning in Android apps.

4.6.2.1 Hard-coded Origins

Overall, we found 145,243 hard-coded HTTPS connections and 760,171 hard-coded HTTP connections in our data-set. For the HTTPS connections, we collected further information such as the number of connections that connect to the same origin and the origin's X.509 certificate whenever possible.

The 145,243 TLS connections we found included connections to 11,203 different TLS-enabled remote origins.

To investigate whether pinning is the recommendable validation strategy, it is important to know if an origin is shared between multiple apps authored by multiple developers or

	Con.	Apps	Conservative	Optimistic
Shared Origin	99,996	40,691	-	-
Unique Origin	45,247	11,547	✓	✓
	145,243	45,549		

(a) HTTPS Origins in Custom Code.

	Con.	Apps	Conservative	Optimistic
Internal Intent	294,846	81,040	-	✓
Public Intent	14,599	8,268	-	-
Parcel	5,729	2,883	-	-
UI Component	31,266	18,766	-	-
Resource	124,356	32,113	-	✓
(Shared) Preference	87,051	31,975	-	✓
Variable	432,985	119,406	-	✓
JSON	96,438	32,200	-	✓
	1,973,040	326,651		

(b) Dynamic Origins in Custom Code.

Table 4.4: Origins in Custom Code – Connections marked with ✓ can be pinned.

used by apps of a single developer only (cf. Section 4.4).

Shared Origins. 1,301 of the extracted 11,203 hosts were present in multiple apps that were authored by multiple developers. We assume these hosts not to be under the control of an app's developer and hence recommend the default certificate validation strategy, since host-ownership is not given (cf. Section 4.4). This affects 99,996 of the TLS connections we identified in our data-set (cf. Table 4.4a).

Table 4.5 lists the top 10 shared origins in custom code we found in our data-set.

Unique Origins. 6,012 origins were unique to one single app while 3,890 origins were included in no more than five apps owned by a single developer. Hence, we assume 9,902 origins to be under the control of a single developer each. We recommend pinning for these apps and their connections, since both host- and code-ownership are given (cf. Section 4.4). This affects 45,247 TLS connections in our data-set (cf. Table 4.4a).

To determine whether leaf pinning or CA pinning is the right choice, we analyzed the deployed certificates for the respective origins. Overall, we gathered 7,941 unique certificate chains. We used Androids pre-installed root CA certificates and Androids certificate validation strategy to verify the validity of the origins' certificates and found

Hostname	**Con.**	**Apps**
graph.facebook.com	220,697	111,559
m.facebook.com	110,903	104,745
www.googleapis.com	30,101	20,120
bugsense.appspot.com	18,402	18,285
www.starbucks.com	32,063	16,029
www.facebook.com	39,969	13,923
docs.google.com	29,240	11,872
mobileclient.paypal.com	39,214	10,963
api.twitter.com	34,641	10,551
svcs.paypal.com	38,175	9,999

Table 4.5: Top 10 Remote Origins.

that 7,177 of all chains could be verified successfully, while verification failed for 764 chains. Of these non-validating certificate chains, 182 certificates were self-signed; 170 certificates were issued by an unknown CA; 335 certificates were already expired and for 160 certificates hostname verification failed. Table 4.6 gives an overview of the chains and the affected connections and apps in our data-set.

Verification Result	**Con.**	**Apps**
Chain Ok	40,433	10,176
Self-Signed	1,966	486
Custom CA	269	81
Expired	1,709	546
Hostname Mismatch	870	258
	45,247	11,547

Table 4.6: X.509 Certificates Statistics.

We recommend pinning for all of these connections (cf. Section 4.4). We recommend leaf pinning for the connections that use a self-signed certificate and CA pinning for all other connections (cf. Table 4.7).

Type	**Connections**	**Apps**
Leaf Pinning	44,978	11,247
CA Pinning	269	81

Table 4.7: Pinning Statistics (Conservative).

4.6.2.2 Dynamic Origins

While the unique origins in custom code are good candidates for pinning, the majority of origins for connections in custom code were not hard-coded into the apps at compile time (cf. Table 4.4). These connections' origins can depend on different external factors such as Intents, UI components etc.

For these connections, we distinguish two scenarios:

Conservative. In the conservative scenario, we assume that connections that use dynamic origins cannot be pinned. This assumption prevents us from over-reporting the applicability of pinning, but probably underestimates its applicability as well. Assuming this scenario, 45,247 of the 1,062,810 TLS connections we found in our data-set would be good pinning candidates, which makes up 4.25%.

Optimistic. In this scenario, we assume some of the 1,973,040 dynamic origins connections eligible for pinning. We still assume that connections that get their input from Public Intents, Parcels and UI Components are no good pinning candidates, since the app developer probably does not have control over the actual origin strings values. That leaves some of the remaining 1,921,446 connections that depend on dynamic origin strings eligible for pinning. We assume that – as for the custom code with hard-coded origins – 16% of all connections are HTTPS connections (cf. Table 4.4), which leaves us with 307,431 HTTPS connections. If we again assume that, like for the hard-coded custom code origins, 31.1% of the HTTPS connections can be pinned (cf. Table 4.4), we can recommend 95,611 connections for pinning. In combination with the 45,247 connections from the conservative scenario, we recommend to pin 140,858 connections in the optimistic scenario. However, while only 86.33% of all known HTTPS connections are implemented in third party library code, but 88.07% of our *assumed* HTTPS connections happen via third library code, the connections we optimistically recommend for pinning make up only 3.8% of all (assumed and definite) HTTPS connections (as opposed to 4.25% for definite HTTPS connections). We can optimistically pin 140,858 connections as opposed to only 45,247 connections in the conservative case. Naturally, we are unable to specify which pinning strategy we would suggest, but extrapolating from the conservative scenario i.e. applying the percentages of which pinning strategy is applicable in the small conservative data-set to the larger data-set, 140,020 cases would be eligible for leaf pinning, while we would recommend CA pinning for 838 connections.

4.6.3 Update Frequencies

In addition to the strategy classification, a crucial requirement for pinning to work properly is the possibility to deploy quick updates for apps to distribute new pins. The applicability of pinning for Android apps in general depends on how quickly developers can push new certificate pins to their users' devices. Hence, we were interested in assessing the frequency for app updates.

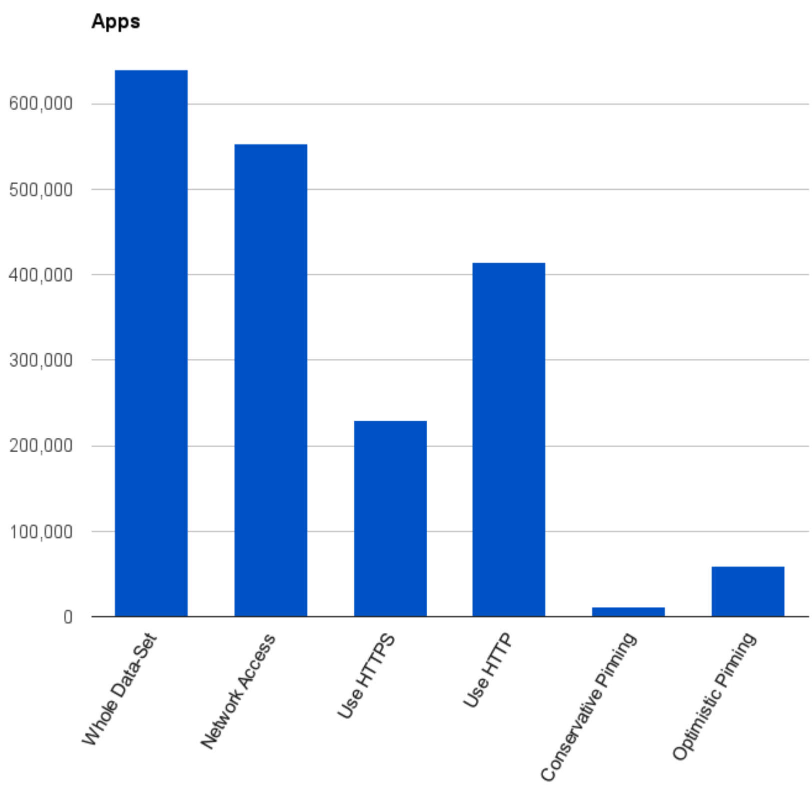

Figure 4.1: Statistics and Classification Results for Apps; *Network Access* includes apps with custom-coded, library- and dynamic HTTPS and HTTP connections.

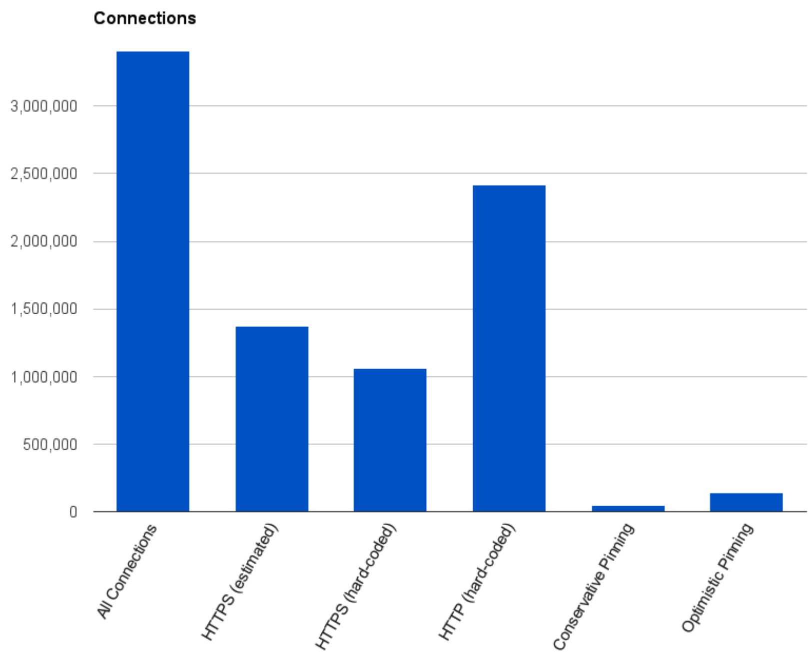

Figure 4.2: Statistics and Classification Results for Connections.

Although newer Android versions have an auto update feature for apps, this feature is opt-in. Furthermore, the default is that app updates are only downloaded in case a device is connected to a WiFi. Hence, even auto-updates are not guaranteed to happen instantly.

Information about update frequencies is not easily accessible via Google Play. To gain insights into users' update behaviour, we cooperated with a popular anti-virus software vendor for Android with an install base of 5 million devices. Our cooperation partner runs a telemetry program and gathers user data for all users that participate in that program. From January 2014 to December 2014, we collected data for 784,721 unique apps and 871,911 unique users. The 871,911 users that participated in the telemetry program and gave their consent to anonymously analyze the data for our research yield the following meta information:

Pseudonym We assigned a 256-bit random pseudonym to each device to protect the users' privacy. The pseudonym did not reveal any private information.

DeviceInfo We collected manufacturer- and device model information as well as the installed Android version.

DeviceFlags We gathered three different flags for every device: (1) Whether developer options were enabled, (2) whether app installs from untrusted sources were allowed and (3) whether USB debugging was enabled.

PackageInfo For every (pre-)installed app we gathered the package name and version code.

PackageHashes For every (pre-)installed app we gathered SHA256 checksums of the packages and their corresponding signing keys.

Timestamps We gathered timestamps for when we saw an app version installed on a device.

Our interest focused on third party apps such as facebook or games, as these apps get updates pushed via Android's default update mechanism. We excluded Google apps and device vendor specific apps from our analysis, since these can be updated through special update channels provided by Google or the device vendor. We identified Google apps and device vendor apps based on the keys they have been signed with [113].

For third party apps, we found that on average 40% of all users update to a new version on the release day. Around half of all app users update within the first week after release and 70% update within 30 days after release. However, to update all devices, on average 200 days elapse. Figure 4.3 illustrates the average apps update periods.

These numbers illustrate that app updates are unsuitable for security-critical app components. In case of pinning, half of all users of an app would be unable to use the app during the first week after the update is released, since the pinning-secured TLS handshakes would fail [2].

[2] We assume that apps use TLS-secured connections for critical components and do not work properly without Internet access.

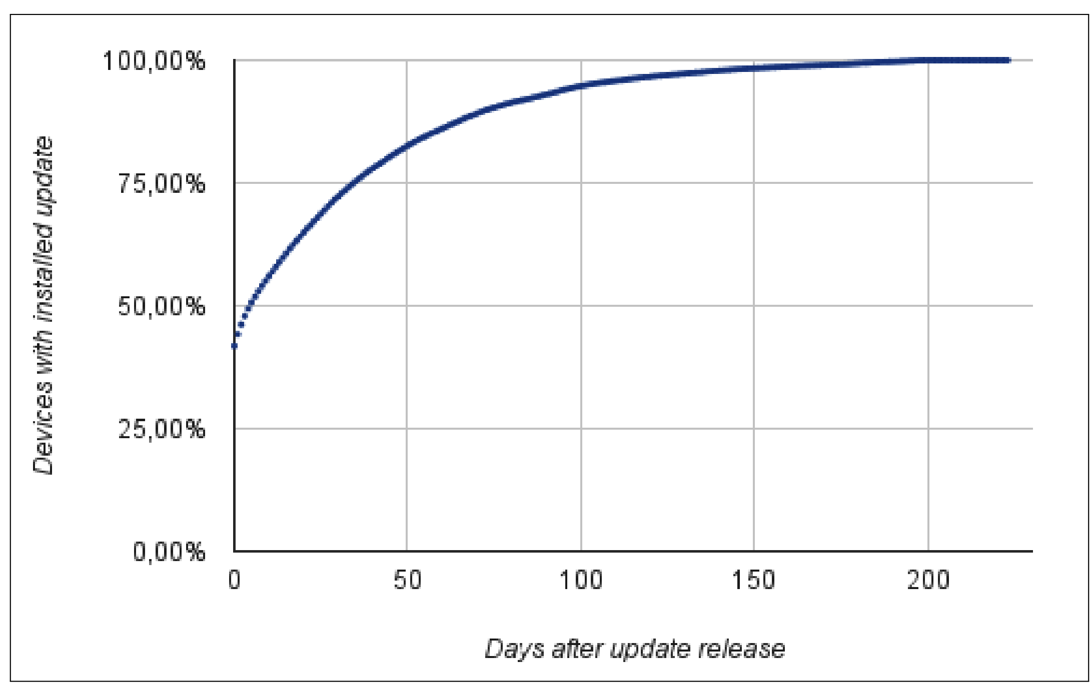

Figure 4.3: Average days between the release of a new app version and its installation.

4.6.4 Discussion

Our results have multiple major implications: The majority of TLS connections happen in third party libraries (86.33%) and another 68.85% of connections are established with shared hosts. However, although only 4.25% of all TLS connections seem to be good candidates for pinning, the current situation of only 45 apps that actually implement pinning leaves much room for improvement. An additional hurdle for deploying pinning are the lazy update frequencies of many users; losing half of the users for a total of one week after a certificate (pin) update is unacceptable for both app developers and their users. To facilitate the use of pinning for eligible developers, we cannot recommend to hard-code certificate pins into app binaries. Instead, pins should be included via configuration files that can easily be updated via a secure remote connection. Such an update mechanism could be enforced immediately and developers would not depend on the (lazy) update behaviour of their users. An alternative pinning deployment strategy would be to use multiple origins for a TLS connection, e. g. having `pin1.example.com` for one pin version and `pin2.example.com` for a second version. Both mechanisms allow for out-of-band pin updates but come with some extra effort on the developers' side.

4.7 Developer Support

After conducting a large scale analysis for real world Android apps and the applicability of pinning, we were also interested in the developers' point of view as the actors who actually implement pinning. First, we collected informative feedback from developers to learn more about their view on pinning. Second, based on their valuable and constructive feedback, we built a tool that supports developers in the process of deciding whether to implement pinning and eases the implementation.

4.7.1 Feedback

We analyzed all apps in our sample set and extracted possible candidates for pinning [3]. For those, we extracted the email addresses of the developers from Google Play, taking care not to create multiple emails for the same recipient, which left us with roughly 3,200 addresses. We emailed a random sample of 500 developers with an introductory email and the plea to provide us feedback on their experience with pinning, or, if they were not the developer, to please forward the email to their app's developer. Our outreach to developers was intended for a qualitative analysis of their feedback and comments to be used as the foundation to build our tool. A quantitative analysis was not our primary focus. We were able to gather 49[4] responses, of which we manually removed 4 who had answered in a nonsensical way or who clearly did not understand English. After we analyzed the 45 responses, we had gained insight into the major problem areas, a saturation of new input was reached and more responses would probably not have provided more valuable insights.

To build our tool with the best possible feature set and usability in mind, we were mainly interested in the following aspects:

Knowledge About Pinning. 15 (a third) of our participants stated they knew what pinning was. We asked them to explain this knowledge, and their replies varied from correct mentions of "custom, self-signed certificates", "reduction of the reliance on intermediate/root certificates, if a intermediate/root gets compromised you don't." and "mobile apps that talk to the same well known server all the time" over "securing the communication between the app and the server without needing to pay to issuers out there" to "i don't know" and confused and/or wrong answers like "when you change the servers and/or certificates more often". We rated 80% of the answers as sensible.

Key Result: Only a quarter of the developers who gave us feedback have a basic understanding of pinning.

[3]Here we made conservative choices to prevent unnecessarily bothering developers.

[4]Given the lack of a platform comparable to Amazon Mechanical Turk for software developer studies, which also means that they donate their time to our research for free, a response rate of 10% seems quite reasonable.

Desired Change: More detailed and critical explanation of what pinning is and how it works as part of the official Android documentation.

Obstacles. Six participants had considered implementing pinning and decided against it. The reasons ranged from "laziness" over confusion to complaints about the complexity and the lack of an "out-of-the-box solution". To the two thirds of our participants who didn't know what pinning was, we showed a short explanatory text [5] and asked them to rate what they imagined could hypothetically keep them from using pinning or convince them to stay with the standard solution. We showed the same set of possible reasons to the developers who were informed about pinning and asked how much these reasons contributed to their not implementing pinning. They ranked "fear of losing users with old app versions / due to hard TLS fails" the highest, followed by "updates required when a certificate changes" and "complexity of the implementation". They said the standard solution was preferable, because "it is easier", they "trust in the existing CA-ecosystem" and "already own CA-signed certificates", but rather not because of "employing several different certificates".

Key Result: Of those who had heard of pinning, 40% had considered implementing it, but discounted it for being unusable or hard to implement.

Desired Change: Provide concrete sample code for the specific use case or app.

Wishlist We received wishes for "good tutorials and programming examples", "example code", "libraries across platforms", a "native Android API", a "test period and simple implementations" and the possibility to "do the same for the web front-end".

Key Result: Developers want better tool support and support in the decision process.

Desired Change: Easy-to-use tool support.

4.7.2 Tool Support

The developer feedback confirmed that more tool support is required and requested. When offering security solutions, we have to keep in mind that developers usually do not have a strong security focus and are not TLS experts, therefore, choosing and implementing secure solutions must be made as easy as possible. To this end, we built a tool that supports developers with implementing secure certificate validation in general; it additionally helps to decide whether pinning is the appropriate strategy. We made a web application publicly accessible at `https://pinning.android-ssl.org/`[6],

[5] taken and adapted from www.owasp.org

[6] The infrastructure this work depends on is not available anymore as well as the framework implementation is no longer functional and not compatible with newer Android versions. Here, we therefore refer to the source code [228].

which we base on our classification framework, the evaluation results and the developer study's results. We chose to implement our tool as a web application, since it is easily accessible and allows to keep the data backend up-to-date.

Developers and app users can upload APK files and have results presented to them in a clear web interface. First of all, the developers need to upload their app's APK file, whereupon the web application conducts all required information extraction steps (cf. Section 4.4) and presents the developer with an overview of relevant API calls and the corresponding remote origins that could be extracted. To increase accuracy, in the next step the developers are asked (1) whether they hold ownership for the relevant API calls and (2) whether they control the TLS configuration for the extracted remote origins (to keep the workflow as simple as possible and not overcharge the developer with unnecessary information, we filter out well known libraries and popular origins). Involving developers into the decision process is especially important in cases where automatic classification might not reveal accurate results, such as for configured origins (cf. Section 4.4).

Finally, the strategy classification is conducted and we present the developer with our recommendation for making their connections as secure as possible. In case the default strategy was selected, we do not recommend the developer to take further action. In case leaf or CA pinning is recommended, the developers are encouraged to increase their app's security by implementing pinning. However, we also inform the developers about the downsides of pinning in terms of updatability of certificate pins (cf. Section 4.6.3). In a last step, the developer is offered support for implementing pinning.

For any given relevant API call and the corresponding remote origin, concrete example code for pinning is generated over the following steps:

1. The remote origin's certificate is fetched and the corresponding pin is computed.
2. A `PinningTrustManager` that uses the pin for certificate validation is generated.
3. Surrounding code that includes the `PinningTrustManager` into the relevant API calls is generated, e.g. for an `HTTPSUrlConnection`, an `SSLContext` is generated that is initialized with the given `PinningTrustManager`.

The developer can then simply include this scaffold into the app and profit from a higher level of security. We asked the interested developers who left their contact information in our developer study to test our web application; 7 participated and gave positive feedback on its usability.

4.8 Limitations

In addition to the limitations described in Section 4.5.5, our work has three more limiations.

First, the update behaviour analysis we conducted for the users that participated in the AV's telemetry program might not necessarily represent the global update behaviour. However, we think that security affine users who install anti-virus software on their devices have a tendency to update their software more quickly than average users. Therefore, update frequencies for the global Android user population might be even worse.

Second, the feedback we got from app developers was based on self-reporting and might be influenced by a self-selection bias. Since we emailed developers who our classification framework had identified as good candidates for pinning, but offered them no incentive for taking part in our survey, we could only work with the developers who responded, leaving us with an opt-in bias. However, this is best practice for getting feedback from developers.

Third, our tool is currently implemented as a web service and a standalone command line tool. In the future, it would be reasonable to include the classification and recommendation process into the publication process in Google Play: An uploaded APK file could be run through the tool and unpinned but pinnable connections could be pointed out to the respective developer or pinned automatically via a library.

4.9 Conclusion

We conducted an extensive analysis on the applicability of pinning as an alternative and more secure certificate validation mechanism for non-browser software. Therefore, we analyzed 639,283 Android apps of which 229,317 (35.9%) use TLS to secure network connections and conservatively recommend pinning for 11,547 (1.8%) of all apps, or 5.0% of the apps that use TLS. This corresponds to 20,020,535 connections, of which 1,062,810 (5.3%) use TLS, of which 45,247 (4.25%) are conservatively recommendable for pinning. Optimistically, including estimates for unclassified connections as well as connections depending on dynamic code loading, we are able to suggest 140,858 connections or 58,817 (9.1%) apps to take pinning into consideration.

This contradicts the common assumption that pinning is a widely applicable solution for making TLS certificate validation in non-browser software more secure. Of the 229,317 apps we analyzed that make use of TLS to secure (some of) their network connections, 203,159 (88.6%) establish TLS connections via third-party libraries.

While we find that pinning is applicable only for relatively few apps and their TLS connections, a nominal-actual comparison illustrates that there is room for improving the current situation, as only 45 of the 11,481 apps that could benefit from pinning actually implement it. To help us understand affected developers and design a solution, we conducted a qualitative study with 45 developers, where we learned that pinning is relatively unknown and often neglected due to usability problems. These results incentivized us to build an easy-to-use web-application to support developers in the

decision making process and guide them through the implementation of pinning for appropriate connections.

Our work concludes with the following take-aways:

Poor Support for Pinning. The Android API lacks sufficient support for pinning: For low-level APIs, developers have to implement their own certificate validation and need detailed knowledge regarding pinning. For higher level APIs, support for pinning is missing entirely.

Limited Applicability. The application of pinning in non-browser software such as apps is very limited: We recommend pinning for only 5.0% of the 229,317 TLS-enabled Android apps we analyzed.

Developer Education. Developer feedback showed that only a third of the developers who could have implemented pinning had heard of it before. Pinning seems to be confusing and developers misinformed. In the future, better developer education is required as well as better developer support.

Security Updates. Our analysis of update periods for Android apps suggests that Android requires mechanisms to quickly deploy security updates in the future (cf. Section 4.6.3).

Pinning Implementation. The current Android documentation recommends to include pinning information at compile time, i. e. the recommendation is to add pins to the source code of an app. However, our analysis of the update behaviour of Android users suggests that developers should not implement certificate pins into an app's binary. Instead, we recommend to set pins via configuration files that are only accessible by the respective app.

4.10 Summary

In this chapter, we investigated to which extent developers of Android apps are supposedly able to implement pinning in their apps. For this we identified and formalized key requirements for pinning to be applicable. Based on this formalization we conducted a large scale static analysis. As a result, we, however, only found a limited applicability for pinning. Reasons for this are most strikingly related to third party ownership of origins or code. While lack of control over code that initiates an HTTP/HTTPS request makes it impossible to introduce pinning, lack of control over respective origins' server configuration makes it infeasible if or at least harder to maintain certificate pins. But even with control over both, code and origins, the effort for maintenance required to keep certificate pins up-to-date on users' devices is still tremendous, especially in case of emergency situations such as lost or theft of private key information that make prompt

reaction and deployment of updated pinning pin indispensable. This effort required, of course, increases relative to the number of installations. With telemetry data from real Android device users, here, we derived an prodigiously long period of time that is required for successfully deploying changes to users which imposes an additional challenge that needs to be tackled. Our study underlines previous works' findings that developers are no security experts as our results show that only a minority actually grasps the concept of pinning. This makes pinning a viable choice most likely only for a rather limited number of developers and apps.

Aligning our results to the state of the TLS ecosystem and strategies for mitigation in regards to the browser landscape shows that our findings for mobile apps are in line with these pertaining to pinning. Here, with the introduction of HPKP, a notion of pinning for server operators was introduced for browsers (cf. 2.4.3.3). However, in line with our results on a limited applicability of pinning in apps, there is only scarce adoption of HPKP [161]. Analogue as the obstacles and challenges that we identified, the configuration and maintenance of HPKP turned out to be similarly cumbersome [162]. Even more, the use of HPKP has been found to cause more severe problems than it provides protections against MitMAs due to CA breaches. In addition, it can even lead to hard TLS fails, making website unavailable for users. Therefore, HPKP as a protection mechanism against MitMAs failed as an effective security measure in browser software. This led to an early deprecation of HPKP and continuing development on improvements for security in the CA-PKI system (cf. 2.4.3.3). This is vastly in line with our findings for the mobile landscape. There is indeed a likewise limited range of situations that constitute compelling arguments for the use of pinning based on the imposed complexity to implement pinning and the burden of ongoing maintenance. This refutes the previous assumption of pinning as a panacea for problems related to the TLS ecosystem. Moreover, developers, apart from limited knowledge of pinning, miss better tool support but also abstain from pinning because of usability problems as well as fear of hard TLS fails for users. Still, however, where applicable, at least a tool as proposed by us could help to prevent dramatic vulnerabilities as uncovered by previous work [114, 126]. Furthermore, since the time of this analysis, support for pinning has been introduced as part of third party libraries like e.g. OkHttp [79]. Later on in 2016, however, flaws have even been found in the pinning implementation of OkHttp [184]. Thus, even with library support, the usage of pinning can still be error-prone. To address such problems, in the ongoing evolution of Android, configuration-driven platform support for pinning is introduced with Android 7 among further security measures for enhancing network security in Android apps. Chapter 6 will feature an in-depth analysis of these. In addition, however, apart from pinning and custom TLS, our results also give hints to the overall adoption of HTTPS in Android apps since we find a majority of extracted connections to still use HTTP (cf. Table 4.2). Pertaining to the observation that large amounts of network operations are issued in shared code and relate to shared origins, we can theorize that developers' general impact on network security is limited as they are not able to enhance security for origins and code that they have no control over.

From this, we will next speculate that developers with more control over network

connections of apps and more power to maintain can have a larger impact in regards to enhancing network security when being able to upgrade to HTTPS. One particular group of apps – namely apps created with so-called Online Application Generators (OAGs) – supposedly features these properties. These services allow their customers to completely externalize the process of app creation. Especially in hindsight to communication with back end servers, OAGs have complete control and thus makes deployment and maintenance of secure solutions including the adoption easier. Therefore, in the following chapter, we will also elaborate on the role of app generators and their security impact especially with regard to network security.

5

App Generators

Externalizing security decisions to 3rd Parties

5.1 Introduction and Contributions

The proliferation of *online application generators* (OAGs) that automate development, distribution and maintenance of mobile apps significantly lowers the level of technical skill that is required for application development. As a consequence, creating platform-specific apps becomes amenable to a wide range of inexperienced developers. This trend that developers with *"little or no coding or software engineering background"* [123] create software with low-code or no-code platforms has become known as *citizen developers* [123, 164] and has recently received tremendous momentum across the industry. Moreover, many OAGs additionally promise to decrease the app's overall development and maintenance costs since they offer functionality for taking care of various tasks across all phases of an app's life cycle. On our path to assess different platform actors' capabilities for enhancing network security, OAGs seem to be predestined as they relief developers from nearly all responsibilities as related to make security decisions and make app development far more convenient.

However, this convenience comes at the cost of an opaque generation process in which the user/developer has to fully trust the generated code in terms of security and privacy. A large body of literature has revealed various security problems in mobile apps, such as permission management [116], insecure TLS deployment [114, 238], misuse of cryptographic APIs [106], and inter-process communication [82]. These flaws could be attributed to poorly trained app developers that implemented application features in an insecure manner. With the increasing use of OAGs the duty of generating secure code shifts away from the app developer to the generator service. This leaves the question whether OAGs can provide safe and privacy-preserving default implementations of common tasks to generate more secure apps at an unprecedented scale. However, if they fail, their amplification effect will have a drastic negative impact on the already concerning state of security in mobile apps. As of now, the security implications of OAGs have not been systematically investigated yet, and, in particular, their impact on the security of the overall app ecosystem remains an open question: *"Do online app generators have a positive or negative impact on the overall app ecosystem?"*

Our contribution. In this work, we present the first classification of commonly used OAGs for Android apps on various characteristics including their supported workflows, automation of the app development life cycle and multi-platform support. We proceed by showing how to uniquely fingerprint generated apps in order to link them back to their generator. We thereby quantify the market penetration of these OAGs based on a corpus of 2,291,898 free Android apps from Google Play and discover that at least 11.1% of these apps were created using online services. This noticeable market penetration already shows that potential security mistakes and misconduct by OAGs would impact thousands of apps and impose a danger for the overall health of the Android ecosystem.

Analyzing the security of OAGs is non-trivial due to the absence of a documentation of the development process. Instead, these services offer a fully-automated, opaque app

generation process without the possibility to write custom code. App developers have to fully trust that the generated code follows security best practices and does not violate the end-users' privacy. In order to shed light onto the black-box generation process, we perform a comprehensive security audit on apps created by these services using a combination of dynamic, static, and manual analysis. This allows us to document their internal workflow and to discover a new app generation model based on boilerplate code. We then demonstrate that 7 out of 13 analyzed online application generators fail to defend against reconfiguration attacks, thus opening new attack surfaces of their generated apps. We further analyze whether the generated boilerplate code adheres to Android security best practices and whether it suffers from known security vulnerabilities identified by prior research. Our results, both on self-generated apps as well as apps randomly picked from Google Play, suggest that OAGs are responsible for producing vulnerable code including SSL/TLS verification errors, insecure `WebViews`, code injection vulnerabilities and misuse of cryptographic APIs. Within our data set, all analyzed application generators suffer from at least one of these vulnerabilities, combined affecting more than 250K apps on Google Play. Finally, we have a dedicated look onto the services' infrastructure security. Online service-generated apps are typically bound to their providers' backend servers, e.g. for license checks when the service charges a monthly fee. In addition, some services even provide complete user-management modules that require connections to backend servers. Any of this functionality requires a secure client-server communication since either sensitive (user) data is exchanged or configuration files for the boilerplate code is transmitted. However, our analysis reveals that many services build on an insecure and vulnerable infrastructure, e.g. by using insecure TLS server configurations, mixed usage of HTTP/HTTPS and usage of outdated TLS libraries.

We conclude with a thorough discussion of our findings, including potential alleys worth pursuing in future research on generating secure code for such module-based app builders. In summary, we make the following tangible contributions:

- We present the first classification of commonly used OAGs for Android, accounting for various characteristics such as the supported workflows, automation of the app development life cycle, multi-platform support, and their boilerplate-based app generation model.
- We show how to fingerprint generated apps and how to quantify the market penetration of OAGs by classifying 2,291,898 free apps from Google Play: at least 255,216 apps (11.1%) are generated using OAGs.
- We derive OAG-specific attacks, such as reconfiguration and infrastructure attacks and show how these services fail in protecting against these attacks.
- We conduct a comprehensive security audit to show that boilerplate code generated by any of the analyzed services violates security best practices and contains severe security vulnerabilities. To estimate the real-world impact, we validate our findings on real, generated apps on Play.

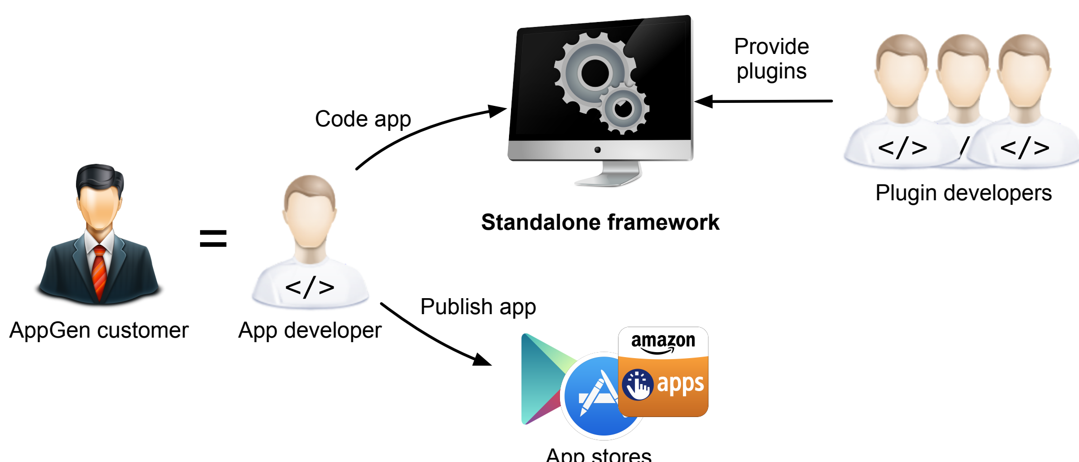

Figure 5.1: Software development workflow with standalone frameworks.

Outline. This chapter is organized as follows. We give a general overview of mobile app generators in Section 5.2 and a classification of commonly used OAGs in Section 5.3. We describe the methodology of our security audit in Section 5.4, present new, OAG-specific attack classes in Section 5.5 and analyze known security issues in Section 5.6. Finally, we thoroughly discuss our findings in Section 5.7, before concluding in Section 5.8.

5.2 Overview of Mobile AppGens

Application generators are tools for partially or even completely automating app development, distribution, and maintenance. The advantages of using application generators are manifold. First, they enable developers to abstract away from implementation aspects and to instead focus only on the conceptual behavior of the application in terms of high-level functionality. Second, they provide functionality beyond core app generation, including support for app compilation, app dissemination, and distribution of patched versions. Third, they offer support for making an app equally applicable to multiple competing architectures, such as Android and iOS. Finally, they may even provide support for recurring, extended app functionality such as user management, user login, and data submission to back-end servers. In this section, we give an overview of commonly used AppGen types within the Android ecosystem based on their supported workflows. Our investigation resulted in three distinct categories of application generators: *standalone frameworks*, *online services*, and software development services that we dub *Developer-as-a-Service* as explained in the following.

5.2.1 Standalone frameworks

Standalone Frameworks (SAFs) constitute tools that offer a core set of abstract application functionality which are then refined by additional code from the app developer. These frameworks typically expect a program written in a platform-independent language as input, e.g., JavaScript and HTML, or C#, and then package user-provided code together with an execution engine into a native app. Many of those frameworks offer plugins that provide commonly used functionality (e.g., in-app browsers or advertisement) or even skeletons for entire apps, which are provided by plugin developers. While these frameworks assist in the creation of an app, they offer little to no support for further phases of an app's life cycle such as app dissemination and distribution of patched versions. To date, these unsupported tasks are typically performed by the app developer. Prominent examples of standalone frameworks are *Xamarin* [307], *Apache Cordova* [30], *PhoneGap* [30], and Titanium SDK [35].

Security responsibilities: In this setting, in contrast to the traditional workflow, not only the app developer is responsible for adhering to security best practices and avoiding security vulnerabilities, but also the plugin developer carries responsibility for the security of offered plugins. Figure 5.1 depicts the developing workflow of SAFs. While SAFs aim at lowering the entry barriers for mobile app development for developers with some programming background, prior research and incidents show that some of these require additional developer action to adhere to safe defaults or offer switches that allow to deactivate these undiscerningly without further warning. One such example is the Titanium SDK. Here, in 2012, Fahl et al. [115] uncovered that iOS apps created with the Titanium SDK have secure certificate validation logic turned off by default with developers having to activate it on their own. [275, 2] In their interview study with developers, Fahl et al. [115] found that developers were not of this in order for their app to adhere to safe defaults. Likewise, a plugin for PhoneGap/Apache Cordova featured a switch for easily deactivating the correct default certificate validation logic in Android and iOS as a mean for allowing self-signed certificates during development. [235, 1] This, however, led to the danger of developers forgetting to remove for production. Both cases already indicate While targeted at non-experts, these settings introduce severe risks and dangers that can massively harm network security of apps and render them vulnerable to MitMAs. Therefore, we cannot derive a pre-assumption as to having impact regarding enhancing adherence to security best practices and network security as for these, developers mostly remain in charge for making security decisions. On this, we already elaborated in Chapter 4.

5.2.2 Online Services

Online services or online application generators (OAGs) enable app development using wizard-like, point-and-click web interfaces in which developers only need to add and suitably interconnect UI elements that represent application components (e.g., email or login forms, in-app browser, QR scanner, social networking widgets, etc.). There is no

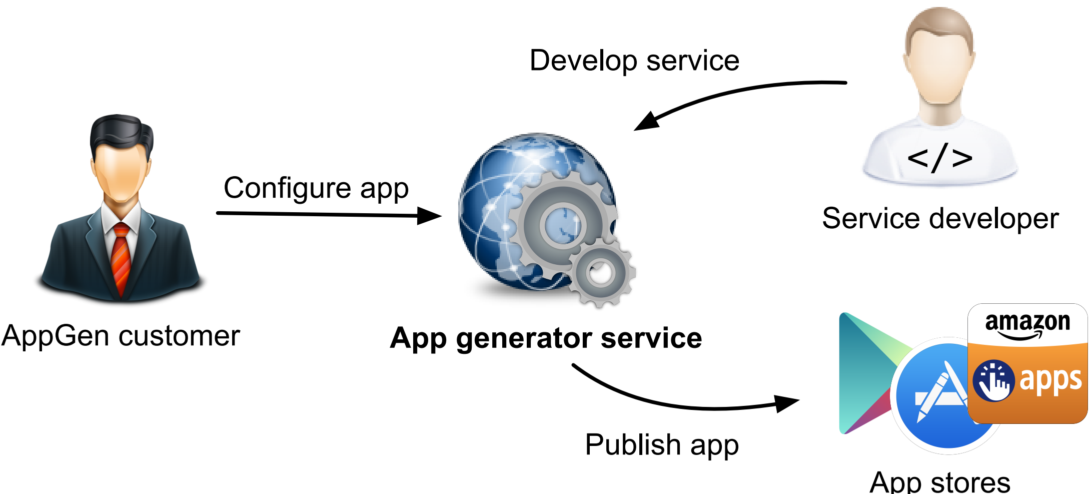

Figure 5.2: Software development workflow with online services.

need and typically also no option to write custom code. For some of these components, they may even provide the necessary infrastructure such as user management, user login, and data submission to back-end servers maintained by the service provider. These online service tools are thus accessible even for laymen developers that lack any prior experience in app development. Moreover, online services offer support for automatically distributing apps over popular channels such as the Google Play Store. In addition to extensively supporting core tasks of the software development life cycle, online services provide business intelligence and analysis features such as audience reports and dashboard for analytics. Prominent examples of online services are *Andromo* [29] and *Biznessapps* [67].

Security responsibilities: Figure 5.2 gives an overview of the app creation process. In the setting of an online service application generator, the customer usually relinquishes any kind of control over the produced code and outsources the app generation to the online services. Thus, the online service and by extension its service developers are responsible for implementing a code generator that produces apps adhering to best practices and without security vulnerabilities. Even more, app functionality might also depend on an OAG's server infrastructure. Hence, assuring network security is also a principal responsibility of OAGs. Prior work has not yet featured an in-depth analysis on this development model except for Acar et al.'s [9] Poster which constitutes preliminary work to what we will present here (cf. 3.2.2). In the remainder of this work, we will therefore investigate the role of OAGs in the Android ecosystem and their security impact.

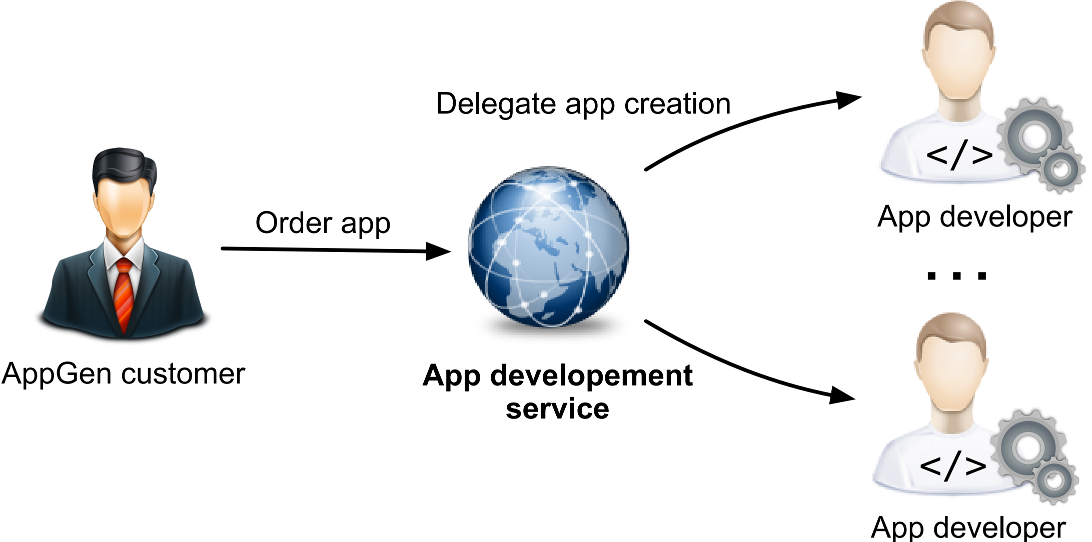

Figure 5.3: Software development workflow with Developer-as-a-Service.

5.2.3 Developer-as-a-Service

Lastly, Developer-as-a-Service (DaaS) does not even expect developers to contribute to the technical development of an app. Instead, the developer rather acts as a customer that orders the whole app creation from a contracted app development service, which then delegates the app creation to a team of app developers that are expected to develop the application based on a set of explicitly spelled out customer requirements. Such requirements are typically collected over the phone or via emails. Widely known examples are *CrowdCompass* [96] and *QuickMobile* [242], services specialized to create customized event and conference apps.

Security responsibilities: In many aspects this setting is similar to the traditional workflow in which app developers create an app. Figure 5.3 serves to illustrate this. However, in order to serve a large number of customers, such app development services commonly use internally a custom, home-brewed app development frameworks and templates to better streamline the production of apps. Thus, the app developers of the development service are primarily responsible for ensuring the security of the produced app. Likewise, we already elaborated on developers in Chapter 4.

5.3 Online App Generators

This section presents a classification of apps of commonly used OAGs and an analysis of market penetration and characterization of OAG-generated apps.

Table 5.1: Classification and fingerprinting of online services sorted by category and app count. For each AppGen, we found multiple distinguishing fingerprint features, but, in all cases, a single feature is already sufficient to uniquely classify an application generator.

	Classification				Fingerprinting Features						
Online Service	Freeware	Multi-platform	Components	Publishing	Package Name	Code Namespace	Files in Package	File Content	Sign. Cert.	Sign. Cert. Subject	Market (# of apps)
Seattle Cloud [254]	○	●	●	●	○	●	●	○	●	●	60,314
Andromo [29]	●	○	●	○	●	●	○	○	●	●	45,850
Apps Geyser [48]	●	●	●	◐	○	○	○	●	●	●	29,190
Biznessapps [67]	○	●	●	●	○	●	○	○	●	●	27,130
Appinventor [39]	●	○	◐	○	○	●	○	○	○	○	25,338
AppYet [51]	●	○	●	○	○	●	○	○	●	●	15,281
Como [91]	○	●	●	◐	●	●	○	○	●	●	10,894
Tobit Chayns [278]	●	●	●	●	●	●	○	○	●	●	8,242
Mobincube [215]	●	●	●	○	●	●	●	○	●	●	8,074
Appy Pie [50]	●	●	●	◐	●	●	●	○	●	●	4,445
Appmachine [43]	○	●	●	◐	●	●	●	●	●	●	4,409
Good Barber [132]	○	○	●	◐	●	●	●	○	○	●	3,622
Shoutem [259]	○	●	●	●	●	●	○	○	●	●	2,638
App Yourself [34]	○	●	○	●	●	●	●	●	●	●	2,273
Mippin [212]	○	○	●	◐	●	●	●	○	○	●	1,785
Apps Builder [47]	○	●	●	◐	●	●	●	○	●	●	1,191
Appmakr [44]	●	●	●	◐	●	●	○	●	○	○	1,058
appery.io [37]	○	●	○	○	●	●	●	●	○	○	846
Apps Bar [46]	●	●	●	◐	●	●	●	○	○	●	700
Mobile Roadie [214]	○	○	●	◐	●	●	●	●	○	○	581
App Gyver [32]	○	●	○	○	●	●	●	●	○	○	386
Appconfector [36]	○	●	●	●	●	●	○	●	○	○	337
Rho Mobile [250]	●	●	○	○	●	●	●	○	○	○	216
Appsme [49]	●	○	●	●	○	●	○	○	○	○	158
App Titan [33]	○	○	●	◐	●	●	○	○	○	○	152
Applicationcraft [42]	○	●	●	○	●	●	○	●	○	○	100
Paradise Apps [232]	●	●	●	◐	○	●	●	○	○	○	3
Eachscape [105]	○	○	●	●	●	●	○	○	○	○	3

● = yes/applies; ; ◐ = applies partly; ○ = no/does not apply

5.3.1 Classification

We used Google search queries to identify a rich set of common application generators. More concretely, we simulated a user who is searching for an application generator using search terms including but not limited to: "app maker android", "android generate app", and "{business, free, diy, mobile} (android) app {generator, creator, maker}". For each result, we selected the first five entries after removing duplicates. We also issued queries to online resources that offer technical and popularity reviews of application generators such as Appindex [38], Werockyourweb [304], Quora [303] , and Businessnewsdaily [258]. We excluded non-online-services and application generators from our analysis, when we were not able to meaningfully assess their market penetration, i.e., we could not determine whether any available app was generated using these particular application generators, see Section 5.3.2.

We have classified all application generators along four dimensions: freeware, multi-platform support, components, and publishing, see the columns on "Classification" in Table 5.1.

Freeware. Some application generators can be freely used (●), while others require a monetary investment (○).

Multi-platform support. While traditional app development requires developers to write distinct apps for each mobile platform like Android, iOS, and Windows Mobile, many application generators allow developers to develop for one platform and then automatically generate "native" apps for additional platforms (●). We write (○) if this multi-platform functionality is not provided.

Components. Many application generators offer supplementary components for common tasks such as ads, app analytics, crash reporting, and user management. We write (●) if features can be conveniently added via simple web forms, e.g. by means of checkboxes; (◐) if users have to rely on visual programming interfaces to add and remove features; and (○) if supplementary components are not offered.

Publishing support. Conventional app development requires developers to write code, compile, and sign an APK, and then distribute it to their users. While writing code, compiling and signing an APK is arguably a smooth process using dedicated IDEs (e.g., Android Studio), app distribution usually requires further manual effort: register a Google Play account (or an account for an alternative market), upload the app, add description text and publish. Some application generators offer to automate this complete chain from producing and signing an app to publishing it on one or multiple markets (●) , while others only automate parts of this support chain(◐) or do not offer support at all (○).

5.3.2 Fingerprinting Application Generators

Once we established and classified our set of online services, we aimed at quantifying the *market penetration* of the individual application generators. To do this in a meaningful manner without relying on bold marketing claims, we identify the number of Android apps generated by the individual application generators. To this end, we first identified unique features of application generators as fingerprints, and then used these features to classify a large corpus of 2,291,898 unique free Android apps. We collected these apps from the Google Play Store between August 2015 and May 2017 using a crawler. The crawler starts with a set of URL seeds and subsequently follows the recommendation links to explore the store. Our crawler revisits previously found apps once per day and downloads them only if new versions are available. Our analysis considered only the latest version of each app.

5.3.2.1 Features

We extracted unique features of application generators using differential analysis between a *baseline* app[1] and sample apps from each application generator. We then manually reverse-engineered all sample apps and created a diff between our baseline app and the sample apps based on the components of a typical Android app. To create this diff, we first analyzed the composition of an Android app, and then identified all parts that can be used to tell apart generated apps and manually developed apps. Our analysis resulted in the following four features (the latter two can be sub-classified, totaling six distinct *Fingerprinting Features*, see Table 5.1):

App Package Names. The first distinctive feature of apps is the app package name. Package names are unique text strings that are used by Google Play to unambiguously identify apps. Application generators often use patterns for generated apps that in turn can be used as a distinctive feature, e.g., `com.Tobit.*` or `{com|net}.andromo.dev*`.

Code Namespaces. Java code is organized in namespaces and similar to package names, application generators may use particular namespaces that we can leverage for our classification. Andromo apps, for example, include code namespaces that contain the substring `.andromo.dev` or the prefix `com.andromo`. A similar example are Tobit Chayns apps, which include code namespaces with the prefix `com.Tobit.android.slitte.Slitte`. In contrast, Apps Geyser apps include code namespaces with the prefix `com.w*`, which is not suitable for classification purposes due to its ambiguity (see Section 5.3.2.2 for further details).

[1] We used the default Android IDE Android Studio to create a single-activity "Hello World" Android app that did not include any external third-party library

Signing Keys. Before uploading an app to Google Play, APKs must be digitally signed. This is a security mechanism to ensure that app updates are distributed by the same entity (e.g., developer). A single key is often used to sign multiple apps, e.g., Seattle Cloud uses a unique key to sign all its apps. We can use this single-key pattern to fingerprint the application generator. Whenever application generators use distinct keys, we can still use further information about the certificate to fingerprint the app, e.g., if all keys have the same subject. AppYet apps, for instance, all share the same certificate subject `/C=CA /ST=ON /L=Oakville /O=AppYet /CN=www.appyet.com`.

Files. In addition to an app's code, apps include a list of files such as images, CSS, or configuration files. These files can be used for the classification as well. For example, AppyPie apps include the file `appypie.xml` in the `assets/www` app folder. We moreover use file content for the classification, e.g., we identify AppsGeyser apps, by verifying whether the elements `<webWidget>` and `<registeredUrl>` of `res/raw/configuration.xml` contain the URL `appgeyser.com`.

5.3.2.2 Methodology

We start our classification by extracting the aforementioned features from our set of sample apps. We discovered that for each AppGen there are multiple distinguishing features that allow to link the app back to its generator. We further found that in all cases a single feature would already be sufficient to unambiguously determine the originating service. The overall classification of application generators with respect to their features is depicted in Table 5.1 in the "Fingerprinting Features" columns.

As an orthogonal investigation, we also fingerprinted apps that have *not* been created by means of application generators. In this analysis, we considered the two major platforms for Android app development: Eclipse ADT [13]—support ended in August 2015—and Android Studio [211].We manually investigated apps developed with both IDEs similarly as described for application generators. Our analysis revealed that Android Studio apps can reliably be identified based on the files' structure. Apps developed and compiled with Android Studio include a folder `res/mipmap` that stores launcher images. This folder structure was introduced in Android Studio 1.1 [133].In contrast, we could not find any reliable identification feature for Eclipse ADT apps. To avoid false positives, we limited our analyses to application generators apps and Android Studio apps we could reliably identify.

5.3.3 Market Penetration

Our app corpus currently consists of 2,291,898 free apps from Google Play. We managed to successfully classify 255,216 (11.14%) of these apps using our feature detection as generated by OAGs. This is a lower bound based on the OAGs we could reliably identify, since OAGs not considered in our classification might be responsible for additional apps.

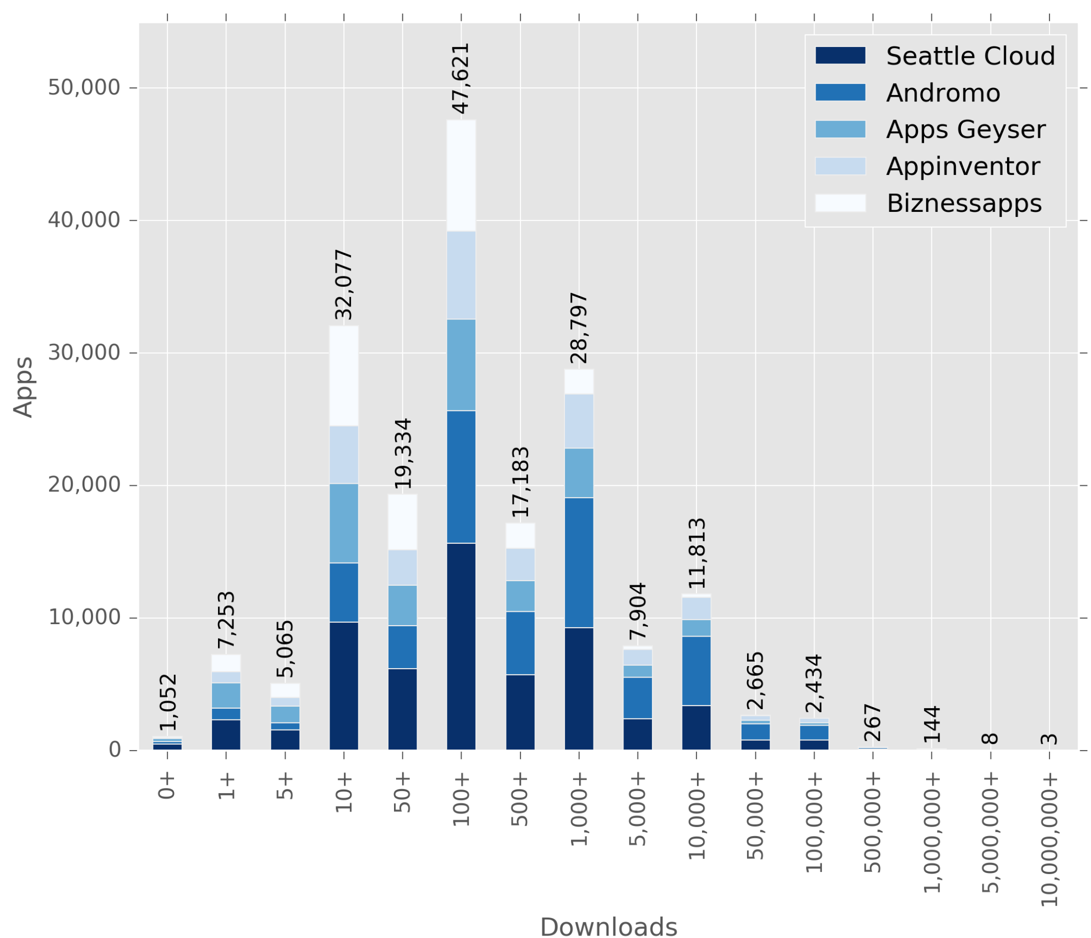

Figure 5.4: Download counts for all apps from the top 5 OAGs using buckets provided by Google Play.

Based on meta data from Google Play, these generated apps account for more than 1.14 bn downloads. Detailed app counts per online service can be found in the *market* column of Table 5.1. The numbers suggest that the majority of OAGs is responsible only for a small fraction of the set of generated apps. In contrast, the five most popular OAGs account for 73% of all generated apps, i.e., 8.12% of our corpus. While the many OAG-generated apps have a small user base, there is also a larger number (>17k apps) with a significant number of at least 10,000 downloads (see Figure 5.4). Reasons for this distribution of download counts originate in the limited set of functionality offered by these services as opposed to traditional app development. In the following, we characterize OAG-generated apps based on the offered functionality.

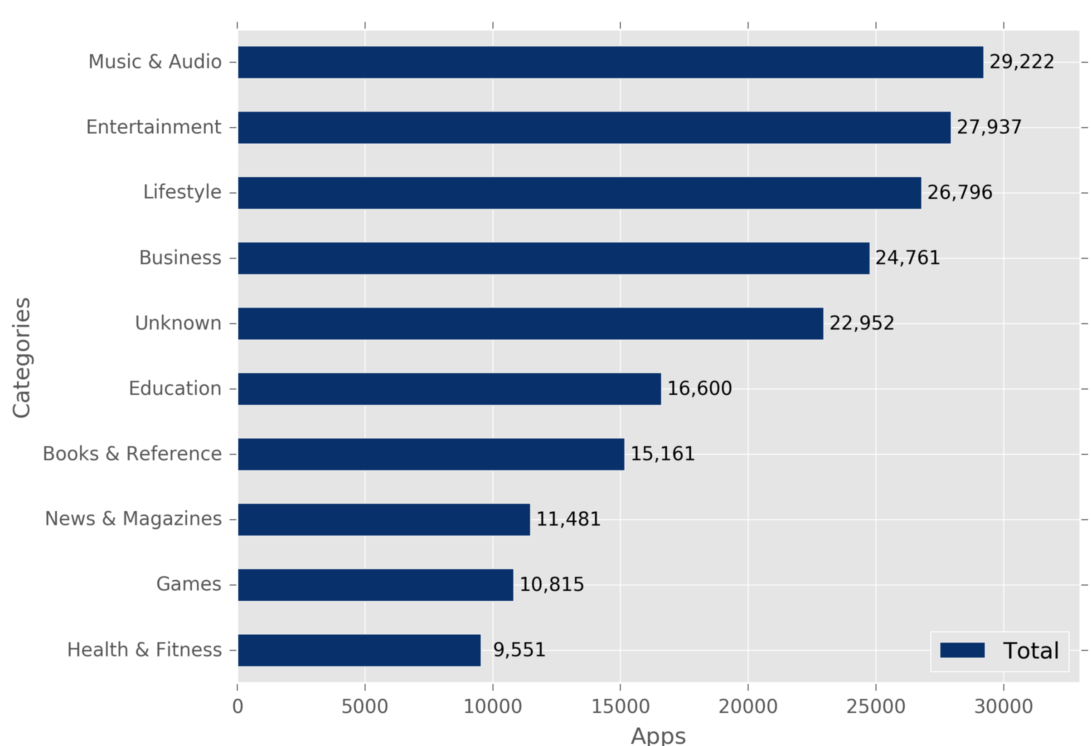

Figure 5.5: Distribution of OAG generated apps by Google Play store categories. The *Unknown* category is for apps that are not classified by the Google Play store

5.3.4 OAG App Characterization

To better understand the potential complexity of the application logic that can be implemented with OAGs, we inspected online services' development IDEs to count the number and types of app components that can be connected to implement the indented logic. We call *component* an app element with a UI and that implements a specific task or functionality. For example, components can be simple UI elements (e.g., buttons, forms, views) or complex modules/plugins (e.g., QR scanners, calendar views, and login forms). The number of components varies across OAGs and it ranges from 12 of AppYet to 128 components of Seattle Cloud. This variety indicates the level of customization that each OAG offers. We also observed a great variety of components. For example, Biznessapps offers 48 components all suitable for business apps, e.g., membership management, mortgage calculator, and loyalty program management components. Another interesting example is Seattle Cloud which provides general-purpose components including simple UI elements such as menu items and image areas, as well as complex ones such as complete PDF readers and barcode scanners.

This variety is also reflected in the different app categories on Google Play. The app category is a string value in the app metadata retrieved by our crawlers that identify

the type of app [2].The ten most popular categories of OAG-generated apps are shown in Figure 5.5. These categories cover apps with non-trivial logic such as *Business* (e.g., document editor/reader, email management, or job search apps), *Entertainment* (e.g., streaming video apps), and *Games*. These categories contain all of the top nine OAG apps with 5M+ downloads. An interesting aspect is that different OAG dominate different app categories, thus suggesting product specialization. For example, the most popular category of Seattle Cloud is *Entertainment* with 11,073 apps, whereas the most popular category for Biznessapps is *Business* with 9,030 apps.

5.4 Analysis Methodology

We now focus on the security of apps generated with OAGs. The naive approach to analyze the security of our dataset is to test all apps systematically. However, we observe that OAG apps are produced in a streamlined process in which app developers do not contribute with source code. We thus hypothesize that the apps share an OAG-specific *boilerplate code* and, as a consequence, either all generated apps share a vulnerability or none. Accordingly, we first identify the app generation model for each OAG, before we analyze the security of the boilerplate code on both self-generated apps and apps from Google Play.

5.4.1 Boilerplate App Model

To identify boilerplate apps, we first build the ground truth with self-generated apps for different OAG. To this end, we select the thirteen most popular OAGs based on their market penetration in Table 5.1 and register as a customer. Among these, six online services (*Appmachine*, *Apps Builder*, *Biznessapps*, *Como*, *Seattle Cloud*, and *GoodBarber*) required us to pay a subscription fee to be able to generate apps and to rent backend resources, such as user management. For each online service we create the same three custom apps to test whether targeted code is generated depending on the selected modules or whether boilerplate code with an app-specific configuration file is output. We create the three custom apps as follows:

- **App1:** The first app constitutes the minimal app, i.e. the smallest app that can be generated in terms of functionality. In most cases, this app just displays a “Hello World” message to the user.
- **App2:** Builds on the minimal app but additionally performs web requests to a web server we control. We perform both HTTP and HTTPS requests to analyze the transferred plaintext data and to test whether we can inject malicious code and emulate web-to-app attacks [127, 83, 203].

[2] A complete list of categories and descriptions can be found at [84]

- **App3:** The third app implements either a user login or a form to submit user data to our server (if the application generator provides modules for such functionality). We chose such common functionality as it is supported by the majority of AppGens and because handling user data usually requires special care in terms of security. In our test set, *Appinventor*, *Biznessapps*, and *Como* did not offer such additional modules.

To test whether an AppGen generates boilerplate code or module-dependent code we analyze the bytecode file(s) of App1–App3 as well as 10 randomly selected apps from Google Play for each service. In the majority of cases, it is sufficient to compare the hash value of the *classes.dex* bytecode file to show that different apps have the exact same code. If the file hash differs we compute a Merkle hash tree over the class hierarchy including package, class and method instruction information to quickly estimate the code overlap (thereby following a similar approach to the one to detect third-party libraries in [57]). The results confirm our observation in which apps generated with OAG are based on a common boilerplate code. In particular, we can derive two distinct generation techniques: monolithic boilerplate apps with configuration files and module-dependent boilerplate code (see Table 5.2).

5.4.1.1 Monolithic Boilerplate Code

All but two online services generate apps with the exact same application bytecode, i.e. apps include code for *all* supported modules with additional logic for layout transitions independent of what the app developer has selected. Apps only differ in a configuration file that is either statically included in the apk file or dynamically loaded at runtime. Some OAGs support both options, e.g. to deliver app updates as config file updates without the need to change the apk file. Further this allows to bind the app to the service providers' backend servers. In this scenario, apks only need to be updated when the online service changes its boilerplate code. Boilerplate apps can be sub-categorized into pure native applications, i.e., these apps include dex bytecode and optionally libraries written in C/C++ (see category A.1), and hybrid applications combining bytecode with HTML/JS (A.2). For HTML/JS apps, bridge code is generated to interact with the Android middleware, while HTML and Javascript is used to render the apps' user interface in a `WebView`.

5.4.1.2 Module-dependent Boilerplate Code

Two online services, Andromo and Appinventor, generate module-dependent boilerplate code for apps, i.e., only the boilerplate code for modules enabled/added by the app developer is stored within the apk file. Hence, apps share the same code for individual modules, but the set of enabled modules might differ. Module-dependent code requires a more complex app generation process for the online service provider, however, the

Table 5.2: OAGs grouped by generation model: Monolithic boilerplate apps with static/dynamic configuration files and module-dependent boilerplate apps.

A. Monolithic Boilerplate Code		
	Static config	Dynamic config
A.1 Native application		
Apps Builder	● (red)	● (red)
Appmachine	○ (green)	—
Biznessapps	—	○ (green)
GoodBarber	● (red)	● (red)
Mobincube	● (red)	◐
A.2 HTML + Native app		
Apps Geyser	● (red)	—
Appy Pie	● (red)	● (red)
AppYet	◐	—
Como	● (red)	● (red)
Seattle Cloud	● (red)	● (red)
Tobit Chayns	—	○ (green)
B. Module-dependent Boilerplate Code		
Andromo		
Appinventor		

● (red) = static config in plain / dyn. config loaded via HTTP
◐ = static config decryptable / HTTPS downgrade
○ (green) = static config encrypted / dyn. config loaded via HTTPS

generated apk is tailored to the configured modules and the chosen layout. The app semantics are not controlled by a pre-defined configuration file. While *Andromo's* app assembly is close to those of the other AppGens, *Appinventor* builds on GNU Kawa and offers, besides modules, a kind of visual programming to give the app developer the choice to implement if-then-else conditions and loops on a high-level. This gives the app developer a bit more freedom in customizing the application but has a slightly steeper learning curve for citizen developers.

5.4.2 Security Audit

Following the initial analysis of the app generation model, we then conduct a security audit on the generated boilerplate code. To this end, we follow a dynamic-static approach. We leverage dynamic testing to monitor the test apps' runtime behavior, e.g., obtain traces of the contacted domains during execution and check the possibility of eavesdropping or modifying those connections. We complement our tests with static analysis (e.g., control-flow graphs, program slicing, backtracking) to overcome the limitations of dynamic analysis, e.g., code coverage. Similar to the analysis of the boilerplate app model, we start our analysis with the self-generated apps of Section 5.4.1. To remove any bias from our set of self-generated apps, we cross-validate our findings with 10 randomly selected apps of the same OAG (=130 apps in total), drawn from our Google Play app corpus. Finally, as the configuration of apps can be provided dynamically by OAG's service, we extend our analysis to the OAG backend servers and the client-server communication.

Our analysis identifies two new attack vectors which are specific to the OAG generated apps, specifically reconfiguration and infrastructure attacks. We present these attacks in Section 5.5. We then test the boilerplate apps for well-known security issues such as code injection vulnerabilities and insecure `WebViews`. The results of this analysis are presented in Section 5.6.

5.5 OAG-specific Attack Vectors

Given the inherently different generation model as compared to traditional app creation, we now describe new OAG-specific attack vectors—Application infrastructure attacks that apply to all OAGs and app reconfiguration attacks that apply to OAGs with monolithic boilerplate code only—and illustrate weaknesses that we found during our security audit.

5.5.1 Application Reconfiguration Attacks

In our set of tested OAGs (see Table 5.2), 5/11 services use either static or dynamically loaded config files exclusively, while the remaining six OAGs use an hybrid approach. Dynamically loaded configs have the advantage that apps can be updated on-the-fly without having to download an updated apk file from an app store. Changes are instantly pushed onto the end-users' devices on startup. However, two AppGens—Tobit Chayns and Biznessapps—do not persist their config locally and thus require a permanent Internet connection to work. As an example, Biznessapps—a paid service—uses this as a license enforcement mechanism, i.e., app functionality is disabled via the config file as soon as the app developer no longer holds a valid license. In general, we found that these configuration files carry *any* app-specific data, potentially including the entire business logic of the app and secret credentials. Hence, there is a strong incentive to carefully protect this file in terms of integrity and confidentiality.

5.5.1.1 Static config files

In our test set, 7/9 static config files are stored in plain and can be read and modified without effort. Only AppYet encrypts its config file, however, at the same time, the passphrase is hard-coded in the bytecode and is identical for every AppYet app. This allows us to write a simple encryption/decryption tool to read and modify the AppYet config files. The Appmachine config was the only one where we are not able to extract information. Appmachine is built on top of Mono for Android, thus, most of the app code is compiled to native code, including the classes that process the non-human-readable config file [3]. As for the integrity protection, we found that in the majority of cases standard Android APIs are used to read these files directly from assets or the raw directory. By inspecting the disassembled app code we can, however, not find any additional integrity checks or obfuscation logic for these config files. This allows, in all but one case, to trivially extract the config file from an apk. Cloning or repackaging apps becomes an easy task, since no code has to be reverse engineered, only the app features and properties within the config file–typically a `.xml` or `.json` file—have to be understood once for each application generator.

5.5.1.2 Dynamically loaded config files

Further, we found that eight out of eleven OAGs (A.1+A.2) load config data dynamically at runtime. All but Biznessapps and Tobit Chayns use a hybrid model of static and dynamic config loading. In five cases, the config is requested via HTTP and transmitted in plain without any protection. Only three OAGs load the config via TLS by default. For Mobincube, however, it is possible to downgrade the request to HTTP. Moreover,

[3] While we abstain from reversing this config file, we assume that, with enough effort, it should be similarly possible to extract information.

none of these AppGens uses public key pinning to prevent MitMAs. Similar to static configs, the complete absence of integrity checks and content encryption allows tampering with the app's business logic and data. This is fatal, since in our tests we could on-the-fly re-configure app modules (enabling, disabling, modification), disable advertisements (for free AppGens), compromise app data (localized strings, about information) or modify the application's sync URL (AppYet). Only few settings cannot be compromised when app content is no longer retrievable from the server due to license expiration (e.g. we found that 5,137 out of 27,130 Biznessapps in our app set already expired).

Having the full control over the app's data and business logic as a network attacker allows a range of different attacks to be mounted with moderate effort. Targeted phishing attacks may allow stealing user data/credentials. On-the-fly replacement of API keys for advertisement may allow an attacker to steal ad revenue. Or an attacker may simply try to deface the application which in turn affects the reputation of the app developer.

5.5.2 Application Infrastructure Attacks

A large fraction of app generators bind their customers to their web services, e.g., for user management or license validation. Particularly, hybrid apps that make use of web technology, like `WebViews`, to deliver the app's logic, bind the generated app to the generator service's web infrastructure. For instance, in our set of AppGens, Seattle Cloud, Biznessapps, and Como bind both their clients and end-users to their service's infrastructure, e.g., by managing the client's user data or by delivering the client app's content to a generated boilerplate app. Thus, the attack surface of the generated app inherently increases beyond the app and its network connection to the web service backend. Consequently, when considering that a single service's infrastructure can serve many hundreds or thousands of generated apps, it is paramount that the app generator service not only follows best practices, such as correctly verifying certificates, but also that the service's infrastructure maintains highest security standards for their web services. Of particular interest is here, whether such services are resilient to remote attackers, i.e., against state-of-the-art attacks against TLS [102, 216, 54, 12, 66, 98] that affect content delivery either to generated apps or app data to the service.

We extract the domains of the different services' backend servers from the generated apps and use available online analysis sites (e.g., Qualys SSL Labs [265]) to check the TLS security of respective backend servers. This particularly includes checks for trusted and valid certificates, support for outdated and weak ciphers and protocols, resilience against recent TLS vulnerabilities, usage of weak keys, and checks whether any domain contacted by default by generated apps is known to distribute malware (e.g., using Google's SafeBrowsing [151]).

The results of analyzing the communication with the server backend are alarming. First of all, only Tobit Chayns and Biznessapps use encryption consistently for any

communication with the backend, while Apps Geyser completely abstains from secure communication (i.e., HTTP only) and the other services secure their communication only partially. For instance, both Seattle Cloud and Como send sensitive data from user input forms like a login form completely in plain text. Moreover, only three services use a valid and trusted server certificate, while, for instance, Appy Pie uses a self-signed server certificate and Mobincube uses a certificate that expired seven years ago. From a cryptographic point of view, all of the services are running an outdated version of SSL libraries that are prone to one or more recent attacks such as POODLE [216], BEAST [102], LOGJAM [12], or FREAK [66]. Mobincube's server was even vulnerable against all of the tested SSL vulnerabilities.

5.5.2.1 Data leakage and Privacy Violations

OAGs typically offer modules to connect to third-party services, like Google Maps or social media platforms like Facebook and Twitter. These modules include code to connect to these services via service-specific APIs. Using these APIs typically requires an API key (and secret). Since OAGs do not create third-party accounts on behalf of the application developer, those AppGens provide their own API key (and secret) to any app created by its service. Some keys require a fee for business/volume usage, like Google Maps keys, hence it is of interest if the OAG protects these keys from (easy) eavesdropping. The combination of leaked Twitter key and secret, e.g., hard-coded in boilerplate code of Biznessapps, allows to send arbitrary authorized requests and in particular to tamper with the account that is shared across all apps generated with this AppGens. Although those keys are application-only authentication keys with limited access rights, the Twitter developer documentation recommends that these *"should be considered as sensitive as passwords, and must not be shared or distributed to untrusted parties."* [41] We found keys and secrets for various different third-party providers unobfuscated in config files, hardcoded in boilerplate code, in the AndroidManifest file, and even in the strings.xml. All identified keys were exactly the same across all analyzed apps, underlining the security impact of boilerplate apps. We could not find a single attempt to obfuscate or protect these keys/secrets.

Besides paid-only services, a large number of AppGens provide their service for free. Similar to normal app development they use different approaches to monetize their apps, such as advertisement and/or tracking. Since the literature has shown that such third-party libraries often leak sensitive user data [155, 270], we especially checked outgoing app traffic and compared our findings with the privacy policies provided by the online service. The results suggest that none of the web domains that the tested apps contacted during our analyses was known for distributing malware. However, four application generators (Como, Mobincube, Biznessapps, and Appy Pie) clearly exhibited questionable tracking behavior. Apps generated with Mobincube sent more than 250 tracking requests within the first minute of execution. In addition, Mobincube includes the third-party library *BeaconsInSpace* to perform user tracking via Bluetooth beacons without the user noticing it. Although *BeaconsInSpace* strongly recommends updating

the privacy policy of apps using their library, we could not find any information in Mobincube's terms and conditions. Appy Pie apps contacted Google Analytics, Appy Pie's backend, and Facebook for tracking. Apps generated by Como automatically registered with different tracking websites, including Google Analytics, Como-owned servers, and others. Biznessapps sends device identifier and location to their backend servers on app launch. While such extensive tracking behavior is already questionable for the free services of Appy Pie and Mobincube, one would certainly not expect this for paid services like Como and Biznessapps.

5.6 Evaluating Known Security Issues

In addition to the specific attack vectors of online services discussed in Section 5.4, we further analyzed the generated apps' boilerplate code for violations of security best practices on Android [141] and vulnerabilities identified by prior research on Android application security and privacy [7], e.g., testing the apps' device-local attack surfaces, such as unprotected components. Again, we used our set of self-generated apps as well as the set of generated apps from Google Play for cross-validation. Whenever feasible we run static tests against the entire set of generated apps from Google Play. Table 5.3 provides an overview of the security analysis results, which we discuss in more detail in the remainder of this section. We distinguish apps in vulnerable, non vulnerable, and risky. We say that an app is vulnerable (●) when we successfully exploit the flaw. We say that an app is risky (◐) when an exploitation scenario exists, but we did not (or could not) reproduce it. Otherwise, we say that the app is not vulnerable (○).

5.6.1 Best-practice Permission Usage (P1–P3)

Apps may request more permissions than actually needed [116, 53], which unnecessarily increases the privileges of third-party code, such as ad libs, that have been shown to actively exploit such inherited privileges and to exhibit questionable privacy-violating behavior [154, 70, 262, 270]. The Android security best practices also explicitly recommend developers to request as few permissions as possible to conform to the principle of least privilege.

Moreover, Android apps are by design allowed to engage in inter-component communication (ICC [109]). However, apps that (unintentionally) export their components for access by other applications, but with no or only insufficient protection, may leak privacy-sensitive data or security-sensitive methods to unprivileged attacker apps [311, 202, 153]—a scenario also warned about in the security best practices. In addition, for certain components, such as `Activities` or `BroadcastReceivers`, the app developer has only very limited means to identify or authorize the sender app [60]. This opens the opportunity for `Intent` spoofing attacks [82, 227] and confused deputy attacks [116], where a vulnerable component acts on behalf of an ICC message from an

Table 5.3: Categorization of considered attack vectors against generated apps in the Android ecosystem.

Attack vector	Free service							Paid service					
	Andromo	Apps Geyser	Appinventor	AppYet	Mobincube	Appy Pie	Tobit Chayns	Appmachine	Biznessapps	Seattle Cloud	Como	Apps Builder	GoodBarber
P1. Overprivileged Apps	○	●	○	●	●	●	●	●	●	●	●	●	●
P2. Unprotected Components	○	●	○	●	●	●	●	○	○	●	●	●	●
P3. Intent Spoofing and Confused Deputies	○	◐	◐	◐	◐	◐	◐	◐	○	○	◐	○	○
P4. Cryptographic API Misuse	○	●	○	●	○	●	●	○	●	●	○	○	●
P5. TLS Verification Errors	○	●	●	●	○	●	○	○	●	○	○	●	○
P6. Fracking Attacks	○	●	○	●	●	●	●	○	○	○	○	●	○
P7. Origin Crossing	●	●	○	●	●	●	○	○	○	●	●	●	●
P8. Code Injection (native / WebView)	○/●	○/●	○/●	○/●	●/●	○/●	○/●	●/○	○/●	○/○	○/●	○/○	○/○

● = vulnerable; ○ = not vulnerable ◐ = risky

attacker app.

5.6.1.1 Security analysis

To detect whether an application is overprivileged, we identify the permission-protected API calls in the application (using PScout's [53] and Axplorer's [59] permission maps) and derive from those the set of required permissions. We complement this list with `ContentProvider` and `Intent` permissions necessary for the app to run properly. We then compare the resulting set with the set of actually requested permissions in the application's manifest. If the latter one is a strict superset of the former one, we call the application overprivileged.

We further check applications for explicitly exported `Activity`, `Service`, and `Content-/Provider` components or potentially accidentally exported components (e.g., by setting an intent-filter without manually setting flag `android:exported to false`). If any of those exported components is not protected with a permission with at least signature protection level, we consider this app as exposing an unprotected component. To also detect receivers potentially prone to `Intent` spoofing attacks, we conduct the same analysis as above for `BroadcastReceivers` , but additionally considering receivers registered dynamically at runtime via the app's context.

5.6.1.2 Results

All AppGens that generate monolithic boilerplate code create over-privileged apps by design (P1 in Table 5.3). As long as an app developer chooses a subset of modules (from the set of 12–128 modules across AppGens), the resulting app has, with a high percentage, more permissions than actually necessary. For instance, the simple *Hello World* app (App1) has between 7–21 permissions for monolithic boilerplate apps, including camera access, write/read contacts, audio recording, Bluetooth admin, and location access. At the same time App1 of Andromo and Appinventor—that generate module-dependent code—request only a single permission and three permissions, respectively.

Application generators do not satisfactorily protect generated apps' components from illicit access (P2). Except for Andromo, AppInventor, and Biznessapps, all tested generators failed to protect one or more components that we identified through manual analysis (e.g., using their package and class name) to be intended as internally-accessible only. This can potentially lead to severe implications for the end-users' or app generator clients' privacy. For example, apps generated with the Seattle Cloud or Mobincube service expose unprotected components for an `InternalFileContentProvider` and `AppContentProvider`, respectively, through which an attacker can read all files to which the app's UID has access, including internal files like databases, private shared preferences, or in case of Seattle Cloud the asset file `app.xml` [255] in which Seattle Cloud apps store, among other things, the user accounts and passwords for logging into

the app. We statically searched for this vulnerable `InternalFileContentProvider` in 60,314 apps from Seattle Cloud and found it in 100% of those apps. We also discovered the vulnerable `AppContentProvider` in 7,953 (98.5%) of analyzed 8,074 Mobincube apps in our test set.[4] This underlines the high security impact of application generators that are using vulnerable boilerplate code. We also discovered that unprotected components are not always within the package domain of the application generator service, but can sometimes be traced back to included third party packages, e.g., by Radius Networks or ZXing, and it remains to be determined whether the app generator failed to correctly protect such third party components or whether the design of those components prohibits a more secure integration.

A particularly worrisome aspect are unprotected `BroadcastReceivers` and `Activity` components that might accept spoofed `Intents` from untrusted senders and act upon such received data (P3). Eight of the tested application generators produce code that is prone to such `Intent` spoofing attacks. For example, Appinventor apps react to a fake SMS notification, Mobincube apps can be triggered to interact with the WiFi service, and Appmachine-generated apps have a remote command receiver exposed that forwards received `Intents` unfiltered to a native command for execution.

5.6.2 Insecure Cryptographic API Usage (P4)

App developers might use cryptographic APIs to secure their data on the end-user device. However, the security that cryptographic APIs can actually deliver, strongly depends on the correct usage of the cryptographic building blocks (e.g., adequate block cipher modes, correct salting, etc.). App developers frequently make mistakes when using those primitives [106], such as using ECB mode for encryption, using a non-random IV for CBC encryption, or using constant salts/seeds (see also [134]). The Android security best practices documentation picked up some of these recommendations and, for instance, advises using `SecureRandom` instead of `Random`, initializing cryptographic keys with `KeyGenerator`, or using the `Cipher` class for encryption with AES and RSA.

5.6.2.1 Security analysis

To detect misuse of cryptographic APIs, we re-apply the analysis methodology presented in [106] by leveraging R-Droid [58] to search for usage of cryptographic API methods and then track their parameters. To this end, we focused on APIs in Android's `javax.crypto` package. For symmetric encryption, we analyzed the usage of the `Cipher.getInstance` parameter, where developers are expected to specify a symmetric encryption algorithm, mode and padding—e.g. `"AES/CBC/PKCS5Padding"`. Similar to related work and security best practices, we rated the use of the ECB mode

[4] We believe that the 121 apps without this provider are older, discontinued apps, built or last updated prior to the introduction of the `AppContentProvider`.

of operation as insecure and rated, additionally, the following outdated algorithms as insecure: (3)DES, IDEA, RC4, and Blowfish. Moreover, the use of non-random IVs for CBC mode or in general the use of a static encryption key is rated as insecure.

To use hash functions, app developers are recommended to use the `MessageDigest` class, where the hash function's algorithm can be chosen via a string parameter—-e.g. `"SHA-256"`. App developers can include message authentication codes (MACs) into their apps by using the `Mac.getInstance` API call. Again, MAC algorithms are expected to be passed as a string parameter—e.g. `"HmacSHA256"`. For hash functions and MACs, we consider the use of MD2, MD4, MD5, SHA0, SHA1 and Ripemd128 as insecure.

Regarding sources of randomness, we consider the usage of `Random` instead of `SecureRandom` as insecure; however, usage of `SecureRandom` is also rated as insecure when a static seed is used.

5.6.2.2 Results

Seven out of thirteen tested app generators failed to use Android's cryptographic APIs securely. A rather pathological weakness seems to be using an insecure random number generator. We discovered in five of seven vulnerable AppGens that `Random` values are generated with a static initialization vector, and most frequently used when generating symmetric encryption keys or initialization vectors for CBC-mode symmetric encryption, rendering the generated apps prone to cryptanalytic attacks. For instance, Biznessapps creates predictable session identifiers by concatenating the output of `Random` with the current system time. Additionally, three of those generators relied on the insecure ECB mode for encryption.

5.6.3 Insecure WebViews (P5–P8)

App developers frequently fail in validating TLS certificates correctly [114, 115, 263, 126], making their apps vulnerable against man-in-the-middle attacks. The Android security best practices have a dedicated, extensive section on security with HTTPS and SSL, explaining the pitfalls and their solutions in implementing a secure SSL connection and even providing tools for testing the SSL configurations of apps.

Moreover, web utility classes, such as `WebView`, allow app developers to combine the features of web apps (e.g., platform independent languages) with those of native Android apps (e.g., rich access to the device's resources). However, the access controls that govern web code (e.g., JavaScript from different web domains) and local code (i.e., Android native code) are not properly composed: the bridge code between web code and local code can interact with the Android system with the same access rights as its native host application, but does not enforce the same origin policy on calls from the

web code to the bridge functions, thus opening this dangerous bridge interface to all loaded web code. The security best practices suggest to enable JavaScript only if really necessary, to prevent cross-site scripting. In addition, it warns that bridges between web code and local code should be used only for websites from which all input is trusted, as it allows fracking attacks [127, 83, 203].

This lack of origin-based protection of the JavaScript bridge also opens the door for various origin-crossing attacks. A particularly concerning cross-origin attack is based on the *scheme* mechanism. Schemes allow apps on the device to be invoked through URLs whose scheme part equals the scheme registered by the app. However, any app can register for arbitrary schemes. In combination with `WebViews` this allows for unauthorized cross-origin attacks [293], when the user clicks on a malicious link in the `WebView`, which refers to a local application that might act on the parameters given by the URL.

Lastly, Android's programming model allows app developers to dynamically load code from different sources, such as public application packages, dex files, or the web via `WebViews`. However, if the application does not correctly verify the integrity and authenticity of loaded code, the app becomes vulnerable to be compromised by an attacker that can modify the loaded (or injected) code. This attack has to be differentiated between platform native code (i.e., dex or C/C++) [238] and web code (HTML, JavaScript) [178, 217]. In the former attack, the attacker is able to modify the loaded code, e.g., dex bytecode on the local file system or inject malicious code into the download stream of such loaded code. In the latter attack, the attacker achieves execution of custom JavaScript code within a trusted website in a `WebView` or manages to navigate a `WebView` away from a benign, safe website to an attacker-controlled website. As a result, the attacker can control the web resources within the compromised `WebView` (e.g., to exfiltrate credentials) and further leverage the `WebView`'s host app's privileges to the extent they are exposed through bridge code between host app and `WebView` instance. Android's security best practices strongly discourages app developers from dynamically loading code from outside of their application for the aforementioned reasons and, again, recommends only loading web code from trusted websites.

5.6.3.1 Security analysis

We tracked the parameters of the `HTTPUrlConnection` and `HTTPSUrlConnection` classes respectively. We rated plain HTTP URLs as insecure and HTTPS URLs as secure. Moreover, we investigate the use of non-default `TrustManager`, `SSLSocketFactory`, or `HostnameVerifier` implementations with permissive verification strategies, which we deem as insecure. Additionally, for `WebViews` we search for custom `SSLErrorHandler` implementations in the `WebViewClient` with a permissive or insecure error handling, which we deem as insecure.

We classify `WebViews` that enable JavaScript as insecure when the bridge functions

expose security- and privacy-sensitive functionality, and as secure if those options are disabled or non-critical functionality is exposed. Since apps with target SDK 19 or higher reject mixed content by default, we consider those secure, unless developers used the `setMixedContentMode` method with the `MIXED_CONTENT_ALWAYS_ALLOW` parameter to deviate from the default; in this case, we consider the app's behaviour insecure according to the previously described metrics.

We further investigate the presence and implementation of the `shouldOverride-UrlLoading()` method of `WebViewClient`. We consider the app prone to origin crossing if the `WebViewClient` is missing, i.e., the opening of the URL is deferred to some installed app registered for the URL's scheme. Additionally, we consider the app prone to this attack if a `WebViewClient` is present, but it's implementation of the `shouldOverrideUrlLoading()` defers the URL loading to apps via sending `Intents` with the URL as parameter.

To determine whether apps load external code, we check for API calls to `DexClass-Loader` and subclasses that load code over (insecure) network connections, which we consider insecure behavior. Apps that use the `URLClassLoader` with HTTP URLs are of particular danger. In case of `WebViews`, we consider code injection possible if either the `WebView` uses insecure Internet connections or if the `WebViewClient` is present but does not override the `shouldOverrideUrlLoading()` function (or implements a permissive URL overriding that opens attacker provided links in the `WebView`). In those cases, an attacker can potentially lure the `WebView` to an attacker-controlled website.

5.6.3.2 Results

Six of the tested application generators rely on web technology, i.e., `WebViews`, to display their client's content. Thus, for those app generators it is paramount to prevent untrusted content from being loaded into the `WebView` or securely sandboxing the `WebView's` interaction with the Android system and other installed apps. Out of the thirteen tested app generators, six failed to correctly handle TLS certificate verification errors and accept any self-signed certificate (P5), which also eases the task of an attacker to inject code into a `WebView` by manipulating the download stream (P8). Apps Geyser catches verification errors, defers the decision about the trustworthiness of the certificate, however, to the end-user, who has repeatedly been shown in the literature to be unable to make such trust decisions correctly [117]. At the same time, we found that none of the investigated online services implemented measures to enhance the TLS security, e.g., by pinning the certificate. Similarly, we discovered that only about half of the tested apps correctly limited the scope of navigation inside the `WebViews` or enforce a same origin policy on the loaded web content, thus opening the possibility to navigate the `WebView` to untrusted web resources that deliver malicious code with full access to the JavaScript bridge to native platform code. This is particularly worrisome when considering that almost all of the tested generators with `WebViews` expose quite substantial JavaScript

interfaces and hence enable fracking attacks (P6). For instance, Apps Geyser exposes over 90 JavaScript bridge functions, providing an attacker with all tools needed, such as camera and microphone access, storage access, or `Intent` sending. Mobincube and Appy Pie even exceed this number by exposing more than 100 functions, including methods such as `createCalendarEvent`, `getCurrentPosition`, `getGallery-Image`, `makeCall`, `sendSMS`, `takeCameraPicture`, `uploadMultipleFiles`, or `processHTML`. Further noticeable is that several of the tested generators convert the loading of a custom URL in the `WebView` (e.g., through a crafted link provided by the attacker) to an `Intent` that will be sent by the generated app to other installed apps. This opens the possibility for cross-origin attacks (P7).

5.7 Discussion

We now interpret the key findings of our online application generator study and propose some short and long-term actionable items to improve the current status quo.

5.7.1 Citizen App Developers on the Rise

The first key finding of our study is that citizen developers are indeed a growing phenomenon in the mobile application development ecosystem. As AppGens promise to decrease the app's development costs, more and more organizations are interested in this new development paradigm. Financial reports, already in 2011, expected citizen developers to build at least 25% of new business applications by 2014 [125], with an estimated a total revenues of $1.7 Billion in 2015 and an expected growth of +50% per year [123]. Our analysis is the first to confirm the growth forecast in terms of market penetration for the mobile ecosystem, showing that at least 11% of free apps in Google Play (250K apps) are already generated by *Online Services*.

5.7.2 Pitfalls of the "One Size Fits All" AppGens' Strategy

Online Services provide simple means of creating apps without requiring any knowledge about programming or mobile operating systems. This is achieved by abstracting the implementation task to some kind of drag-and-drop assembly of predefined modules and by limiting the degree of freedom of app customization. Such a "One Size Fits All" strategy led to a new paradigm in app generation, distributing an apk file with monolithic or module-dependent boilerplate code that is statically or dynamically configured with an app-specific config file. While this provides a convenient way to generate and distribute applications for a large number of clients, from a security perspective, this creates new points of failures, that, if not considered carefully, might compromise end-user security or even online service security.

The results in Section 5.4 show that the majority of OAGs that base their business model on monolithic boilerplate apps fail to properly protect config files from tampering and eavesdropping. Only 2 out 8 OAGs in Table 5.2 correctly use HTTPS to retrieve config files. Moreover, we found that none of the services applied certificate pinning to prevent man-in-the-middle attacks. Similarly, only a single service properly protected its statically included config. However, none of the services checked the integrity of the config file during app launch. This opens the door for many attacks such as reconfiguration attacks, ad revenue theft (through replacing API keys), and, in general, changing arbitrary app-specific data.

Boilerplate apps that use HTML/JS for layouting (see category A2 in Table 5.2) are additionally prone to code injection and fracking attacks (P6–P8 in Table 5.3). This is caused by the web-to-app bridge these services use to access the Android API. Due to the boilerplate app pattern, these bridges expose more functionality than typically necessary and/or are not properly protected from being misused.

Following the principle of least privilege, Andromo and Appinventor (see category B) generate targeted boilerplate code based on the modules selected by the app developer. Since their code generation model follows the traditional app development, they are not prone to OAG-specific security problems such reconfiguration attacks. However, the trade-off is an additional code generation effort, when any combination of pre-defined modules must be flawlessly composable. This is why Andromo, with only 19 available modules, is at the lower end in terms of available modules, while Appinventor with its community support offers notable 59 modules.

5.7.3 Amplification of Security Issues

The increasing use of online services has shifted the duty of generating secure code from app developers to the generator service. Users rarely have options to customize or change their application beyond the ones provided by the service. As a consequence, users have to fully trust the service to generate non-vulnerable code and to not include hidden or non-obvious user tracking or data leakage. Particularly worrisome examples include the paid service Como, that performs heavy user tracking although its privacy policy explicitly emphasizes the importance of the security of the users' personal information and Mobincube that silently tracks users via BLE beacons without explicitly stating this in its terms and conditions.

The results of our security evaluation in Section 5.6 suggest that OAG-generated apps do hardly adhere to security best practices and exhibit common app vulnerabilities that have been identified by prior research. Although these findings are in line with coding practices of traditional developers, the worrisome aspect is the amplification effect of online services, putting millions of users and their private data at risk. Another key insight is, as opposed to traditional apps, vulnerabilities in unused boilerplate code can still be exploited when a network attacker is able to compromise the application

config and re-configure or activate app modules with known security issues or when `ContentProviders` can be queried to retrieve internal data (see Section 5.6.1).

We conclude that in the current online service ecosystem the level of security does not depend on whether it is a free or paid service, but rather on the underlying app generation model. For the two module-dependent code generators Andromo and Appinventor we found the least security issues. Particularly for Appinventor this is unsurprising, since it is open-source and does not follow commercial interests. The boilerplate model is not generally insecure, however, from a security perspective, server communication and config protection require a more careful design. This could include certificate pinning for dynamically retrieved configs, obfuscation or encryption of static configs, and integrity checks to prevent unauthorized tampering.

5.7.4 Missed Opportunity for a Large-Scale Security Impact

A patch to the current situation is to inform online services about the discovered security issues to allow them to fix their code generation. We are currently in the process of a responsible disclosure to allow the respective service providers to fix the security flaws. However, while this is a short-term mitigation, it does not address the root cause of these issues, thus not producing more desirable long-lasting effects. In our opinion, OAG services need a thorough investigation from the research community in the way AppGens are built. This investigation requires a solid understanding of the underlying technique in use. Other areas of research from which lessons can be learned or transferred are tailored software stacks. Prior works have shown that the attack surface can be considerably reduced by compile-time [188] and run-time configurations [269].

5.8 Conclusion

In this work, we present the first classification of commonly used online services for Android based on various characteristics and quantify the market penetration of these AppGens based on a corpus of 2,291,898 free Android apps from Google Play to discover that at least 11.1% of these apps were created using online services. Based on a systematic analysis of the new boilerplate app generation model, we show that online services fall short in protecting against reconfiguration attacks and running a secure infrastructure. A subsequent security audit of the generated boilerplate code reveals that OAGs make the same security mistakes as traditional app developers. But in contrast, they carry the sole responsibility of generating secure and privacy-preserving code. Due to their amplification effect—a single error by an OAG potentially affects thousands of generated apps (250K apps in our data set)—we conclude that *Online Services* currently have a negative impact on the security of the overall app ecosystem. But, at the same time, these services are in the unique position to turn these negative aspects into positive ones through spending more effort into securing their application

model and infrastructure from which ultimately millions of users benefit.

5.9 Summary

In this Chapter, we presented the first comprehensive classification of commonly used OAGs for Android and showed how to fingerprint uniquely generated apps to link them back to their generator. We thereby quantified the market penetration of these OAGs based on a corpus of 2,291,898 free Android apps from Google Play and discover that at least 11.1% of these apps were created using OAGs. This hinted at a high potential of impact that these apps could have on security in the Android ecosystem, especially pertaining to improving network security e.g. by adoption of HTTPS. However, using a combination of dynamic, static, and manual analysis, we found that the services' app generation model is based on boilerplate code that is prone to reconfiguration attacks in 7/13 analyzed OAGs. We showed that this boilerplate code includes well-known security issues such as code injection vulnerabilities and insecure WebViews. Given the tight coupling of generated apps with their services' backends, we further identify security issues in their infrastructure. As root cause, we could identify lack of network security or improper implementation of certificate validation logic. Most strikingly, we find that apps that were created with OAGs feature the same kinds of flaws as prior research has already uncovered in normal apps (e.g. [114, 126]). Due to the blackbox development approach, here, developers are unaware of these hidden problems that ultimately put the end-users' sensitive data and privacy at risk and violate the user's trust assumption. A particular worrisome result of our study is that OAGs indeed have a significant amplification factor for those vulnerabilities, notably harming the health of the overall Android mobile app ecosystem. Pertaining to the role of OAGs as trusted third parties and driving force and amplifier for improvement of network and application security, we find that OAGs – so far at the time of our analysis [5] – miss this chance.

Aligning our findings in this chapter to the evolution of network security in the TLS ecosystem and in browser landscape, we can derive that OAGs lag behind in adopting secure networking communication via HTTPS (cf. Section 2.4.2.1). In contrast, in the browser landscape and the overall TLS ecosystem, we find trends towards a steeper increase in adoption of TLS and HTTPS fueled e.g. also by the advent of Let's Encrypt (cf. Section 2.4.2.1). While being in the unique position to make use of their infrastructure as a key component for secure handling of user data at a scale by consistently adopting HTTPS on their server infrastructure and apps, a majority of analyzed OAGs fails in this regard by relying on insecure server configuration and insecure HTTP protocol making it feasible for attackers to exploit. These are root causes for OAG-specific attack vectors (cf. Section 5.5.1). Apart from this, we find OAGs that contain custom TLS code in their apps. Neither of these , however, constitutes a security enhancing implementation of pinning. Instead, we find the same kinds of vulnerable certificate validation logic as already known from prior work [114, 126]. Thus, OAGs are not different in this

[5] Our analysis was conducted in May 2017

regard. In addition, it is particular worrisome as end-users are unable to notice (cf. 2.5) insecurities and developers have no chance to mitigate when they are bound to an OAG's server infrastructure and are not able to use their own server as well as they cannot make changes or enhancements to generates apps but are exposed to vulnerabilities and misconfigurations stemming from OAGs not adhering to best practices. From this case study of externalizing key aspects such as control over network security, we could – besides the chances for a positive amplification effect – highlight corresponding limitations and risks. While OAGs constitute the most extensive case where control is completely externalized to a third party, in this work, we already mentioned other forms of third party code and their capabilities to contribute to enhancing or degrading network security. Alongside, in Section 5.2.1 we also briefly covered SAFs that allow web developers to create mobile apps without further expertise. Here, however, previous work has already covered insecurity pertaining to vulnerable custom TLS implementations that can render apps vulnerable to MitMAs. Aside from that, here, developers are fully responsible for network security in contrast to OAGs. Even more, previously in Section 4.10, we already covered a vulnerability in the implementation of pinning in the OkHttp library. Taking everything into account, in this and the previous chapter we found that both, relying on individual developers to take required actions to increase network security, or externalizing control and responsibility to application generators or even libraries does not necessarily have a widespread positive effect on the overall security and also network security of mobile apps and the Android ecosystem. This is in line with e.g. the case of HPKP (cf. Section 2.4.3.3). The adoption of HPKP was very low as it could cause more harm than providing providing protection as previously discussed. Instead, in the TLS ecosystem and browser landscape global platform-wide measures have been introduced. These either relate to increasing the adoption of HTTPS by website operators or enhancing security of the TLS ecosystem by introduction of ecosystem-wide approaches such as e.g. Certificate Transparency (cf. Section 2.4.3.3) instead of relying on individual website operators, CAs or browsers to protect against adversaries on their own and potentially exposing users to the risk of MitMAs.

Analogous to this, the ongoing evolution of the Android platform and Google Play ecosystem led to similar developments. Instead of relying on developers to adhere to secure best practices and implement these on their own, which turned out to be error-prone (cf. Section 3.1.1), newer Android versions introduce safe defaults and security configuration mechanism among platform policies to improve the state of network security on Android. In the next chapter, we will therefore discuss these measures and assess their efficacy.

6
Platform

Rolling out security as new safe default

6.1 Introduction and Contributions

Studying the security of Android applications has a long history [7] and was heavily influenced by the seminal paper by Enck et al. in 2011 [108]. A myriad of investigations demonstrated that developers struggle with different aspects of implementing Android application security mechanisms correctly [107, 106, 82, 108, 238, 224]. The number of affected users of Android applications vulnerable to different types of attacks due to incorrect security implementations goes into billions [305].

While developers fight with many different security challenges, custom TLS certificate validation security received attention early on in 2012 [114, 115] and has become a hotly debated topic over the years [126, 263, 229, 85, 115, 274, P1, 94, 226]. The problem not only affects Android applications but turns out to be a broader issue in secure programming [160, 65, 19]. 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[P1, 115, 274].

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 6.1). 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 NSC [220] 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 [135, 137, 136]. They prohibit new apps and updates to include insecure certificate validation logic. While previous work (e.g. [P2, 299, 181, 243] 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 [114] and 2013 [115], 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 [21].

While in Chapter 4 and Chapter 5 we already elaborated on developers' and third parties' capabilities to enhance network security as well as their this-regarding impact, the

overall goal of this work is to investigate the current status of TLS certificate validation security in Android apps and assess the efficacy of new platform-wide measures for enhancing network security in the Android ecosystem.

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. [114]. 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 6.3 and 6.3.1.

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 [114, 126]. We replicate work by Fahl et al. [114] 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 [135, 137, 136] and demonstrate the importance of further research in this area. We report and discuss this contribution in Section 6.4.

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.

6.2 Background on TLS and Android

TLS is the most widely deployed network protocol to secure communication channels between clients and servers [101, 249, 16]. 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 [27]. 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 [114] or to protect against malicious activity related to the CA-PKI ecosystem (cf. 2.4 and 2.5). Before the introduction of NSC, developers had to implement custom certificate validation logic using Android APIs [306, 168, 297]. However, using custom code commonly leads to vulnerabilities [114, 126], 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. [114]). 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 [184], affecting both app developers as well as libraries such as OkHttp's CertificatePinner [80] [99].

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 6.1 illustrates important changes in chronological order.

6.2.1 Network Security Configuration

With the release of Android 7, Google introduced the Network Security Configuration (NSC) [220]. NSC supports certificate pinning, custom CA certificates and debugging flags, both globally for all network connections or for specific domains [220].

Figure 6.1 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 [73]. Instead, HTTPS is used by default [143]. Developers can set the `cleartextTrafficPermitted` flag if they want to enable HTTP (cf. Listing B.3 in the Appendix) [222]. Alternatively, developers can configure cleartext traffic support in the application manifest by setting the `android:usesCleartextTraffic`

[1] The entire X.509 certificate validation process is much more complex, but left out here for brevity. We refer the interested reader to 2.4 and [68].

Table 6.1: Chronological overview of TLS-related events in the history of Android:

		Date	Android Version	Description
android	1	2015-10-05	Android 6 (API 23)	Android introduces the "android:usesCleartextTraffic" flag for Manifest files, and removes the Apache HTTP Client library [144, 138, 25].
▶	2	2016-05-17		Google Play blocks apps containing unsafe implementations of the X509TrustManager interface [136].
android	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 [139, 72].
▶	4	2016-11-25		Google Play blocks apps containing unsafe implementations of the onReceivedSslError method in WebViews [135].
▶	5	2017-03-01		Google Play blocks apps containing unsafe implementations of the HostnameVerifier interface [137].
android	6	2017-08-21	Android 8 (API 26)	Android adds support for the "cleartextTrafficPermitted" flag for the WebView class [140].
▶	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) [97, 145] available to those apps.
android	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 [143].
▶	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) [97, 145] 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) [97, 145] 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) [97, 145].

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

attribute [3]. Since Android 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 [110]. 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 [139, 72]. However, developers can re-enable user-installed certificates by setting the `user` flag (cf. Listing B.4), which is a security downgrade comparable to the situation before Android

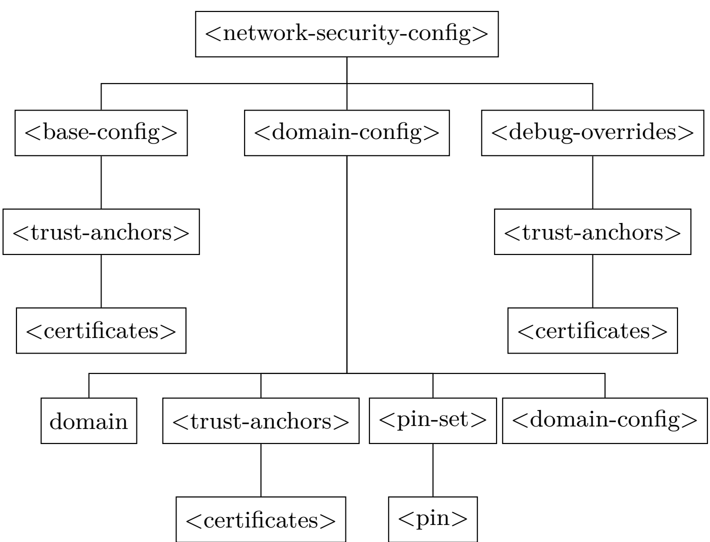

Figure 6.1: NSC files contain `<base-config>`, `<domain-config>` and `<debug-overrides>` configurations, including custom CA (`<trust-anchors>`) and certificate pinning (`<pin-sets>`) configurations. Cleartext traffic can be permitted or forbidden using the `clearTextTrafficPermitted` flag globally for specific domains.

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. [286]

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).

6.2.2 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 [142]. Since August 2018, Google Play has only accepted apps and updates that target Android 8 [97], 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 [145].

6.3 NSC Adoption and Security

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

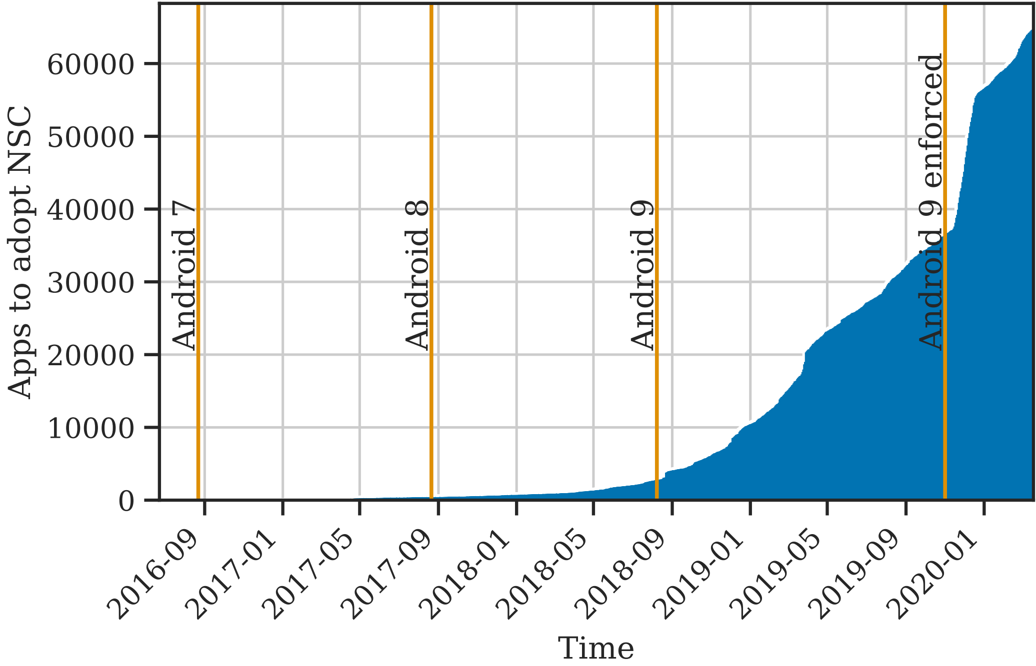

Figure 6.2: Adoption of NSC over time. The release of Android 9 had a significant contribution.

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 6.2) for Android 7. We downloaded the set of Android applications from Google Play using the unofficial Google Play protobuf API [150]. 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 [P2].

[2] The release date of Android 7.

Of the 1,335,322 free Android apps we analyzed, 99,212 implemented custom NSC settings. We used the OBFUSCAN tool [302] 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 6.2: 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. [282]) to implement NSC.

Table 6.2 gives an overview of the target SDKs and custom NSC settings of the apps. Figure 6.2 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 6.3). Even though Android rolled out NSC in September 2016 with Android 7 (cf. Table 6.1), widespread adoption was delayed until early 2019 (cf. Figure 6.2). This correlates with Android 9 introducing HTTPS as the default protocol for web requests in late 2018 (cf. Table 6.1).

6.3.1 Security Analysis of Custom NSC Settings

Below, we analyze the security of custom NSC settings. Figure 6.3 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. [3]

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.

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

[4] Cf. Section 6.2.1 for an overview of all possible NSC settings developers can configure.

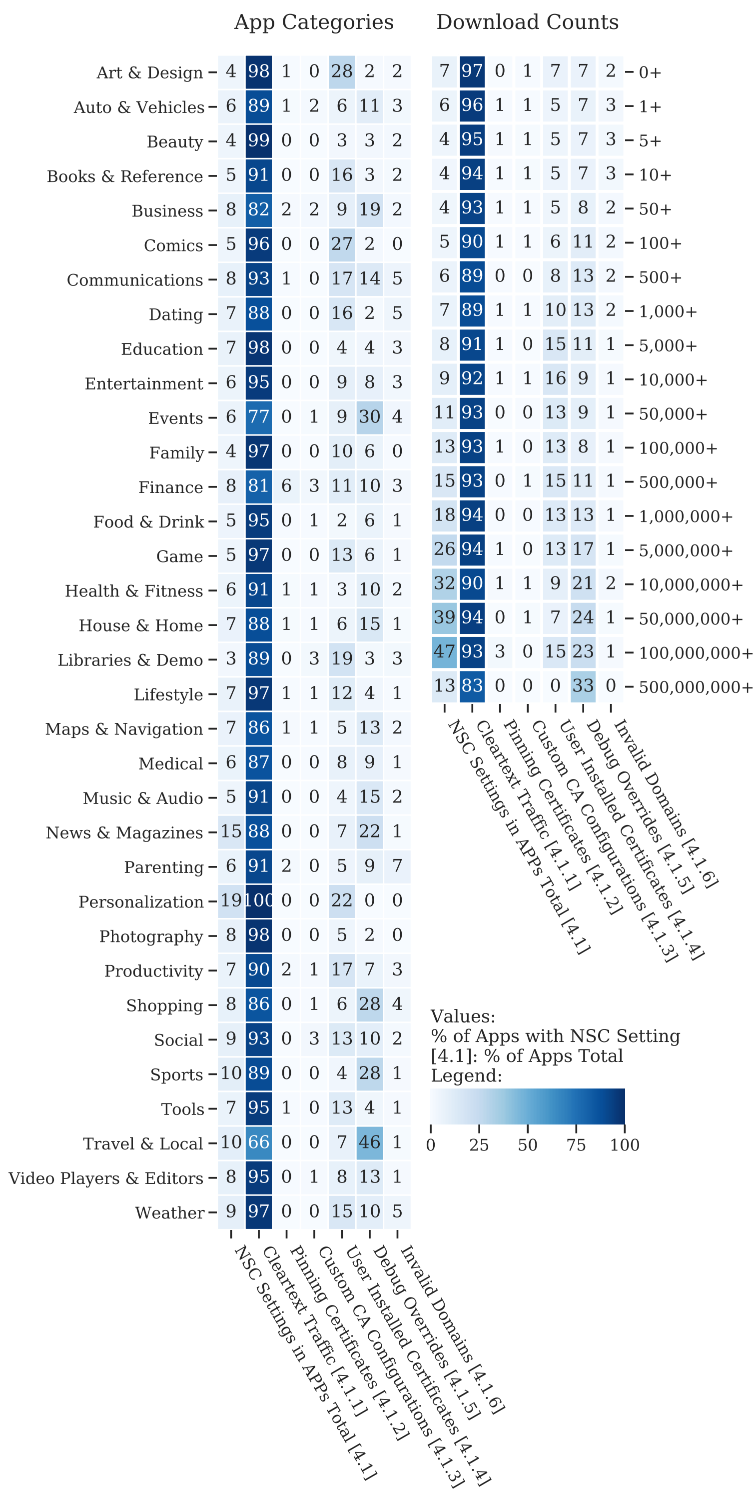

Figure 6.3: Distribution across the features we analyzed, and app categories, and download counts.

Table 6.3: 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. [223])

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 6.3 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 than 50,000 downloads to be more likely to include a custom NSC file (11-47%, cf. Figure 6.3). 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.

6.3.1.1 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 6.1). 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 6.4 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. 6.2.1) 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.
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 downgrade; this affected 24,653 distinct domains. We find valid HTTPS connections for 8,935 applications and argue that downgrading safe defaults was unnecessary. Table B.2 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

Table 6.4: 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

impact. However, they might result from copy & paste from Facebook's cache proxy library that is used in many apps [221] or from debugging configurations developers use for testing.

Table 6.5 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 6.5: 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 B.4 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.

6.3.1.2 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 B.3, 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 B.1 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 [27]. 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 [220]. 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 6.3.1.6. 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 [223]. 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 [220]. 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.

6.3.1.3 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 B.6 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.

6.3.1.4 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 6.2.1).

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 6.3).

Since user-installed certificates increase the attack surface for MitMAs, developers are encouraged to use debug-overrides instead (cf. Section 6.3.1.5).

[5] We only discuss custom CA configuration in production code here. Custom CA configurations for debugging purposes are addressed in Section 6.3.1.5

6.3.1.5 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 [100]. We found 318 apps that register custom certificates in `<debug-overrides>` (cf. Table B.5 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* [81] MitMA proxy tool was the most popular in our dataset and included the substring *"Charles Proxy Custom Root Certificate"*. Contrary to `<debug-override>` configurations, these are used in production code and can therefore pose a security threat. Therefore, the Android documentation discourages their use [27]. While this list is not exhaustive, it shows that developers mis-use NSC settings for debugging purposes although NSC provides distinct debugging options.

6.3.1.6 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 [273, 28, 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 [24] or in the MoPub app monetization documentation [128]. 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 [128] (cf. Listing B.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 [282] and have no further effects, they are likely copied from the Trustkit demo application[283].

6.3.1.7 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 [21], 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 6.3.1.6 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.

6.3.1.8 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.

[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'.

6.4 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 [137, 136].

In particular, Google announced to detect three implementations: TrustManagers vulnerable to attacks using invalid certificates [136], HostnameVerifiers vulnerable to malicious domains and hostnames [137], and WebViewClient.onReceivedSSLError implementations that do not appropriately handle HTTPS errors in a WebView [135]. To investigate root causes for the findings in previous work [P2, 299, 243, 181] 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 6.6 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 [285, 280, 179, 149]. We also distinguish if the faulty code was reachable (R), hidden behind debug options (D), or unreachable (U).

6.4.1 TrustManager Implementations

We started with investigating insecure TrustManager implementations [114, 126].

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 [114, 126] and is frequently discussed in online Q&A forums [177, 176]. Given Google Play's announcement [136], 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`

[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.

Table 6.6: 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

(TM-R-renamed) to match the most common TrustManager name reported by Fahl et al. [114]. This passed as well, which implies 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 [114] and discussed in online developer forums [287] 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.

6.4.2 HostnameVerifier Implementations

Our second set of experiments investigated the efficacy of Google Play's safeguards against insecure hostname verification in apps [114].

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 [114], 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.

6.4.3 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 [P1], 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.

6.4.4 Reproducing Complaints of Developers

Finally, we conducted a set of experiments to reproduce complaints of Android developers we found online [285, 280, 179, 149] 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 [285] in which a developer reports that the use of the Acra [40] 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 [180] library for HTML parsing in version 1.11.1. Developers report that their apps were rejected because of a vulnerable TrustManager implementation [280, 179] (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[20] library providing interfaces for HTTP connections. This library included a vulnerable Trust-Manager in version 1.4.9 [149] (LB-U-asynchttp) and like previous experiments passed successfully without warnings.

6.4.5 Insecure Apps in Google Play

Motivated by the experiments above and previous work [P2, 299, 181, 243], we replicated a study performed by Fahl et al. in 2012 [114]. However, we replaced the outdated MalloDroid tool [114] with the most currently published tool for vulnerable certificate validation logic in Android apps CryptoGuard [243] 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 [243] 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. [114] and Georgiev et al. [126] and show that the Google Play security checks for TLS are inefficient.

6.5 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 [17]. 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 [107, 106, 82, 108, 115, 274, P1, 94, 114, 122, S1, P2]. 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.

6.6 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 [114, 126].

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. [114] and Georgiev et al. [126] 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 [229]. This is in line with findings of Possemato and Fratantonio [239]. 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. 6.3.1.1). We report similar findings regarding configurations for *127.0.0.1* (cf. Section 6.3.1.1) and copy paste behavior (cf. Section 6.3.1.6). 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. [114, 115] and Oltrogge et al. [P1] 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 [239] on the low occurrence of pinning in NSC files and unintentional misconfigurations of pins that occurs across their sample

(cf. Section 6.3.1.2 and 6.3.1.6).

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 [239] report problematic domain usage, e.g. by developers using both the dummy domain *example.com* or invalid parameters for pins or domains (cf. Section 6.3.1.6). 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 [223] 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. [224] 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 [243, 57] to identify the vulnerabilities we tested, our findings and previous work [P2, 243, 299, 181] 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 [243] or LibScout [57] 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.

6.7 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.

6.8 Summary

Android applications have a long history of being vulnerable to MitMAs 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 performed a large-scale in-depth investigation of the effectiveness of these countermeasures: First, we investigated 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 MitMAs. 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. Aligning the evolution of the Android platform to that of the TLS ecosystem and our findings from the previous chapters, we find a range of actions that have been taken to tackle the problems pertaining to the state of network security in Android. Most notably, these constitute the introduction of enhanced enhanced safe defaults. Especially, the enforcement of HTTPS by default which constitutes a major contribution in regards to enhancing network security in Android apps. While we find the new network security configuration approach (NSC) is most dominantly used to allow cleartext transmission via HTTP again, a huge variety of apps profits from the new safe defaults in regards to network security. To this end, there has been a steeper growth in adoption of HTTPS same as in the browser landscape (cf. 2.4.2.1) and in huge contrast to our previous findings in Chapter 5). For those apps that

still opt-in to cleartext traffic again, however, our manual analysis finds major risks imposed (cf. Section 6.3.1.8). Thus, we find these new platform-wide safe default to have a vastly positive impact in contrast to our previous findings on third parties and their dubious impact (cf. Chapter 5). As to custom TLS, the introduction of NSC closely relates to prior proposals (e.g. [115, 274]) and integrates these directly into the operating system, as opposed to our proposal in Chapter 4. Comparing our results, however, we can likewise derive that the scarce application of pinning is in line with our previous findings on the applicability of pinning in Chapter 4 and the use of HPKP in the browser landscape [161]. This confirms our previous conclusion of pinning not being a panacea for the problems and insecurities related to the CA-PKI trust model. Even more, the results of our studies clearly underline that relieving developers and third parties as much as possible from the burden, responsibility and risks pertaining to having to implement network security on their own is more effective than relying on each app's and library's developer to be able increase the security level. While enhanced safeguards and policies for prohibiting insecure implementations were deployed, their efficacy turned out to be too limited to protect against susceptible custom TLS code in apps.

Following the further developments in the overall TLS ecosystem, we find an ongoing transformation (cf. 2.4.3.3). For example, the introduction of mechanisms such as CT has led to protection against MitMAs due mischievous actions of CAs no longer only being a task on the client side but actually being elevated to a platform wide effort mandatory for all CAs. This distributed effort for logging and openly tracking every issuance of a certificate facilitating detection and reaction of misconduct. In addition, although not widely used yet, CAA even aims at allowing certificate owners to only allow authorized CAs to issue certificates for their respective domains. On top of that, the reduction of certificate lifetimes further reduces the impact of potential security incidents. All these techniques serve to illustrate a sustainable shift of responsibility and accountability that make pinning increasingly losing importance as only countermeasure against the risk for MitMAs in non-browser software. To this end, the Android platform – as any system utilizing TLS and the CA-PKI approach – profits from the developments in the overall TLS ecosystem. This ranges from the steeper growth in adoption of HTTPS and the enhanced security level already established with platform-wide deployed techniques. Also, in the light of the experience regarding the deployment and maintenance of pinning in the case of HPKP (cf. 2.4.3.3), CAs also mandate against the use of pinning [271]. Still, as enhancements such as CT, CAA and shorter certificate lifetime do not apply for private CAs and CAA not being mandatory for every domain, there are still viable reasons for choosing custom TLS behaviour in apps. For that, however, our analysis indicates a configuration-driven approach like NSC to be a more viable choice for realizing use cases involving custom certificate validation logic like support for pinning or private CAs as opposed to developers implementing custom TLS on their own or relying on third parties to do correctly, since our studies still feature a significant prevalence of insecure custom TLS code in apps (cf. Section 6.4.5) and limited capabilities of safeguards to ban insecure implementations. However, in fact, even our analysis on the use of NSC as a simpler and safer mean for custom TLS uncovered several obstacles and new pitfalls

in regards to misconceptions that can be harmful. Therefore, in the light of all these findings and on the effect of platform-wide default measures for improving the state of network security in Android apps and the Android platform, we derive that safe default security presets requiring as few developer interventions as possible turn out to be potentially be the most effective strategy for a sustainably higher level of network security.

7

Conclusion

7.1 Summary

The Android platform bases on a browser-based approach for realizing encrypted connections via TLS. Likewise, it gives app developers extensive control and freedom for modification of certificate validation logic. This, however, also abetted a disastrous state of custom certificate validation implementation in Android apps. This state has been uncovered in 2012. [114, 126] Since then, a multitude of efforts have elaborated on this topic (cf. Section 3.1). With this dissertation, we presented a line of research that explores the state of network security in Android apps based on contributions from three publications with the author of this dissertation as first author. [P1, P2, P3] In addition to previous works, each of these works contributed insights in regards to the state of network and app security in Android apps from a different perspective. On top of that, we aligned our observations and findings pertaining to the situation on Android to the overall evolution of the TLS basing on the CA-PKI approach and the browser landscape. The CA-PKI is the paradigm on which most non-browser software including Android also rely (cf. Section 2.5). Due to this, the evolution of network security on Android is also highly dependent on that of the overall TLS ecosystem.

Developers. First, we investigated the role of developers in regards to overcoming obstacles in the widely deployed CA-PKI-based trust model for TLS with the use of pinning as an advanced strategy for protection against MitMAs. We proposed a tool to address the urgent state of vulnerable custom certificate validation implementations in apps. Therefore, we formalized the applicability of pinning in apps and, in addition, assessed its prevalence in a large-scale analysis. However, as a result of our investigation, we find that only a small fraction of cases where developers could be advised to custom TLS code as a means to harden security. Most important reasons include that developers only have limited control over third party origins their app interacts with and code that is part of third party libraries. At the same time, our results already underline the additional efforts for maintenance required on the developers' side which might cause more risk than reward from a network security perspective, even compared to the risk imposed by the standard CA-PKI trustmodel.

Third Parties. Second, we dealt with app and network security in app development from a different perspective by focusing on third parties as app developers – namely Online Application Generators in particular – and their capabilities for security enhancements. The previous observations regarding the limited power of individual developers to introduce enhanced network security and likewise huge risk to cause more harms than alleviation to the hazardous state of the CA-PKI lead to the question if fewer developers with more and wider control over many apps can have a positive impact on security. Therefore, we focused on apps created by app generators in order to assess their amplification effect on app and network security at a greater scale due to mass deployment of shared code in thousands of apps. In theory, their development model could help overcome the obstacles and the overhead related to deployment and maintenance of enhanced network security implementations. To investigate we first analyzed the market

for app generator solutions and estimated their prevalence in the Android ecosystem. Based on that, we conducted a comprehensive security analysis of the most common services regarding their adherence to common security best practices including network security. In our analysis, we found, however, the same sorts of vulnerabilities in apps created with OAGs have as compared to traditional apps. Even more, we could identify even new threat models specially related to OAGs. These attack vectors are strongly related to an insecure use of networking APIs or lack of adoption of secure protocols and still using HTTP instead of HTTPS which, especially, is in strong contrast to the overall steeper increase in adoption of HTTPS in the TLS ecosystem and browser landscape. Thus, we derive that, while being in a unique position to have a positive effect on app security on a larger scale, so far, OAGs miss their chance of a positive amplification effect making widespread implementation and deployment especially in regards to network security more secure. Hence, also completely externalizing app development and security decision making responsibility to third parties did not result in more secure apps and adoption of enhanced measures for network security and certificate validation. Even worse, by employing the same vulnerable practices as already uncovered by prior research, due to the amplification effect of OAGs, their impact can be potential even more dangerous.

Platform. Third, we, again, changed perspective and shed a light on the role of the Android platform and Google Play ecosystem in hindsight to the deployment of security measures and policies to enhance network security on Android. From our previous analyses, we learned that both individual developers and third party providers lack either suitable power or qualification to cause improvements regarding network security. Here, we therefore focused on measures installed at a platform level and analyzed the efficacy of these. These encompass NSC as new security settings mechanism, safeguards and new safe defaults. The introduction of NSC aimed at easing the implementation of custom TLS behaviour such as pinning without developers having to write code, and thus to protect against potentially erroneous code. However, our analysis showed a scarce adoption of NSC. Although helping to relieve from the urgent state of custom TLS code, pinning still imposes a huge management overhead and complexity related to maintaining certificate pins as well as even the configuration-driven approach provided by NSC turns out to be error-prone itself as we find several forms of mis-configuration. Furthermore, we still find only a limited use of pinning which supports our previous findings in Chapter 4. In addition, new developer policies and safeguards target at obstruction of apps that contain vulnerable custom TLS implementations. Our analysis of Google Play safeguards identifies these, however, to only provide sufficient protection against rather common flaws as we find in our penetration testing efforts for the enforcement of Google Play developer policies. Also the results of our replication of Fahl et al. [114] reflect this as we still find a high prevalence for insecure custom TLS code violating developer policies which is are in line with with findings of prior research before the introduction of these measures (cf. 3.1.1). Finally, the new safe defaults and especially the enforcement of HTTPS in apps turn out to be the most promising measure to improve network security at a global scale. With the new safe defaults we find a reduced attack surface for MitMAs in apps by default facilitated by an increased

adoption of HTTPS. Still, we find many apps and their developers to undermine these new safe defaults which also became apparent as most common use of NSC. Even more, our manual study of apps adhering to this practice confirms the severe risks for MitMAs by undermining safe defaults and allowing cleartext traffic transmission – thus the use of HTTP – again. Especially, as our results indicate a huge security degradation due to the broad impact of these settings if used undiscerningly. Taking all these measures into account, we find that all of these aim at improving the state of network security. While we find the new safe defaults deem to be most effective, the efficacy of NSC and safeguards is rather limited and accompanied by degradation of network security.

For each of our analyses spanning over more than five years, we also aligned our findings to the evolution of the overall TLS ecosystem within the same time period. We both find similarities and discrepancies. For pinning as a panacea for the weakest-link property of the CA-PKI, we find similarities regarding use of HPKP on the web and in Android apps in regard to complexity in hindsight to implementation and deployment as well as effort required for maintenance. We find a similarly scarce usage of pinning techniques. Due to low acceptance and huge impacts of mis-configurations on the web in browsers, HPKP was deprecated and removed (cf. 2.4.3.3). However, in Android apps pinning remained as a viable option notwithstanding the dramatic state of custom TLS code. To alleviate, tremendous research efforts proposed alternative approaches for custom TLS (cf. 3.1.2) including ours in Chapter 4 [P1] to avoid risks of erroneous implementations. At last, the Android platform introduced NSC as a mean to keep developers from implementing custom TLS code on their own but still being able to address special use cases. On the other side, in the TLS ecosystem, new developments and innovations have been found. These led to major upheavals of the ecosystem involving all stake holders and especially shifting responsibilities to CAs and giving domain owners more control (cf. 2.4.3.3). In that, also non-browser software such as Android apps indirectly profits from these enhancements as mischievous behaviour can be detected and resolved earlier.

Concerning the adoption of TLS and HTTPS, the situation is vastly similar. Earlier, we summarized the challenging path to wide-spread adoption on the web in browsers (cf. 2.4.2.1). Same as for the problems related to the CA-PKI-based default trust model, major events also fueled a steeper increase of adoption, there were additional driving factors. These most notably constituted organizational and technical upheavals to ease the cost and effort required making deployment TLS feasible. With these, e.g. web standards and other regulations to further enforce web operators to support HTTPS and quit use of HTTP. Since the adoption of HTTPS in Android apps tightly depends on servers' support for HTTPS, there is a coincidentally high prevalence for insecure HTTP connections even for apps created by OAGs that have full control over their infrastructure. However, similar to the adoption of HTTPS in browsers, with deployment of HTTPS slightly becoming default in browsers, the Android platform finally introduces enhanced safe defaults enforcing use of HTTPS without developers being in charge to opt for enhanced network security. While for the vast majority of apps in the Android platform profits from this and feature enhance networks security, using NSC, developers are free to opt-out of these safe defaults. Undiscerning use of these tools can therefore

still cause major harms pertaining to network security and put end users at risks.

7.2 Future Work

Prior research led to manifold discovery of vulnerabilities in software. Here, in the context of this dissertation, the history of Android app development features a wide range of drastic examples for insecurities caused by developers or third parties making security decisions in fields they have no proper expertise for, potentially having bad impact on network security at a massive scale. Chapter 6 identified stricter safe defaults as most effective measure for enhancing network security in apps. The results from our investigations in this dissertation, however, still suggest a need for intensified research on tool and API support for developers as well as more effective safeguards to address problems as still identified in Chapter 6.

The author of this dissertation, therefore, envisions different directions for future research to continue this line of research. Instead of relying on developers and third parties to adhere to best security practices, Android frameworks and developer tools should enforce these as effectively as possible. These safe defaults should align to state-of-the-art in the TLS ecosystem. Therefore, one promising research direction lies in further elaborating on the enforcement of safe defaults as well as on keeping these up to date relative to up to date security practices regarding TLS. The results from our investigations in this work show that the capabilities of developers to have a positive impact on network security and even enforce a higher level of security than already provided by default are vastly limited. Even more, custom code e.g. for certificate validation – regardless of whether by individuals or third parties – poses a great risk to cause even more severe vulnerabilities when deviating from standard settings. Our work therefore strongly suggests that a principal measure for better security lies in transforming the need for manually implementing enhanced security measures into up to date safe defaults making overwriting of default behaviour as non-essential as possible. Enforcement of safe default is therefore a key aspect regarding the improvement of security. Future work could therefore further elaborate on mechanisms to strengthen adherence of developers to safe defaults.

Pertaining to developer support, efforts like e.g. in Chapter 4 [P1], Wei et al. [298] and Nguyen et al. [224] already laid groundwork on this as well as in the Android tool chain, basic LINT features, developer policies and safeguards were implemented. These either try to make recommendations for improving security by e.g. providing code samples or spot vulnerabilities in app code. However, on both sides, we find such efforts to be largely limited either regarding their applicability and soundness of provided solutions (cf. Chapter 4) or in terms of efficacy due to limits of static code analysis (cf. Chapter 6). Therefore, in contrast, integrated in developer tools, there should be better support for developers in adhering to safe defaults such as adoption of HTTPS. Especially, this should also scrutinize developers' motivations to change standard behaviour that might result in the deviation from safe defaults and validate

their necessity and correctness. Chapter 6 illustrated that with regard to e.g. NSC there is a substantial need for such improvement in tool support as current LINT features can only address a mediocre subset of potential problems. The same applies for custom TLS implementations. From this arises another direction for future work focusing on the design of more usable TLS-related high-level APIs complementing NSC as a configuration driven approach. These should be as simple as possible, while being designed powerful enough to address common in order for constituting a genuine replacement for the current complex low-level APIs followed by deprecation and removal of those old APIs at least from the set of interfaces available to app or library developers. Developer studies regarding usable cryptographic APIs for encryption already hint at promising results in this direction to address the still urgent state of custom TLS code [8, 152]. Lastly, in this work we find that security decisions are not under developers' control – in accordance to Possemato and Fratantonio [239] for NSC – and may also not be driven by security requirements that developers aim to implement but instead may be functional requirements e.g. for libraries or third parties that apps rely on. This implies that necessary security downgrades should be at best isolated from the rest of the application in order to limit the security impact compared to our results from our NSC analysis. Although already proposed concurrently to our work but not yet implemented in Android, this is therefore a crucial direction for further research. In contrast to the rather coarse settings in NSC, these should be more fine-grained in regards to code origin, especially concerning library code. Also, as our analysis still revealed an urgent state of custom TLS-related code.

Putting this all together, however, the observations in this dissertation imply that the creation of an ecosystem with safe defaults that at the same time shift away the need from developers for making and implementing security decisions on their own or abolishing these as good as possible. This should reduce the need for developer having to implement security-critical behaviour on their own, leading to the most promising direction for future research that aims at software environments that move the task of network security as far out of the focus of developers as needed to accommodate the risk of urgent effects possibly related to insecure custom implementations.

Bibliography

Author's Papers for this Thesis

[P1] 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)*. USENIX Association, Washington, D.C., USA, Aug. 2015, 239–254.

[P2] 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: *Proc. 39th IEEE Symposium on Security and Privacy (SP'18)*. IEEE, San Francisco, California, USA, May 2018, 634–647.

[P3] Oltrogge, M., Huaman, N., Amft, S., Acar, Y., Backes, M., and Fahl, S. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: *Proc. 30th Usenix Security Symposium (SEC'21)*. USENIX Association, Vancouver, B.C., Canada, Aug. 2021.

Other Papers of the Author

[S1] Fahl, S., Harbach, M., Oltrogge, M., Muders, T., and Smith, M. Hey, You, Get Off of My Clipboard. In: *Proc. 2013 Financial Cryptography and Data Security (FC'13)*. Ed. by Sadeghi, A.-R. Vol. 7859. Springer, Okinawa, Japan, Apr. 2013, 144–161.

[S2] Huaman, N., Amft, S., Oltrogge, M., Acar, Y., and Fahl, S. They Would do Better if They Worked Together: The Case of Interaction Problems Between Password Managers and Websites. In: *Proc. 42nd IEEE Symposium on Security and Privacy (SP'21)*. Vol. 1. IEEE, San Francisco, California, USA, May 2021, 1367–1381.

Other references

[1] *[CB-12809] Google Play Blocker: Unsafe SSL TrustManager Defined - ASF JIRA*. https://issues.apache.org/jira/browse/CB-12809. Last visited: 05/31/2021.

[2] *[TIMOB-6715] iOS: validatesSecureCertificate default value is always "false" - Appcelerator JIRA*. https://jira.appcelerator.org/browse/TIMOB-6715/. Last visited: 05/31/2021.

[3] *<application> | Android Developers*. https://developer.android.com/guide/topics/manifest/application-element#usesCleartextTraffic. Last visited: 09/22/2020.

[4] *1496088 - Certinomis: certificate for test.com, O=Entreprise TEST*. https://bugzilla.mozilla.org/show_bug.cgi?id=1496088. Last visited: 05/19/2021. 2018.

[5] *470897 - Investigate incident with Comodo CA that allegedly issued bogus cert for www.mozilla.com*. https://bugzilla.mozilla.org/show_bug.cgi?id=470897. Last visited: 05/19/2021. 2008.

[6] *About upcoming limits on trusted certificates - Apple Support*. https://support.apple.com/en-us/HT211025. Last visited: 05/19/2021.

[7] 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)*. IEEE, 2016.

[8] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., and Stransky, C. Comparing the Usability of Cryptographic APIs. In: *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[9] Acar, Y., Backes, M., Fahl, S., and Stransky, C. *When Laziness Snaps Back – The Impact of Code Generators on App (In)Security*. In 1st IEEE European Symposium on Security and Privacy, IEEE EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016. Mar. 2016.

[10] Acar, Y., Fahl, S., and Mazurek, M. L. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In: *Proc. 2016 IEEE Secure Development Conference (SecDev'16)*. IEEE, 2016.

[11] Acar, Y., Stransky, C., Wermke, D., Weir, C., Mazurek, M. L., and Fahl, S. Developers Need Support Too: A Survey of Security Advice for Software Developers. In: *Proc. 2017 IEEE Secure Development Conference (SecDev'17)*. IEEE, 2017.

[12] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., and Zimmermann, P. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[13] *ADT Plugin | Android Developers*. http://developer.android.com/tools/help/adt.html. Last visited: 12/01/2017.

[14] Akhawe, D., Amann, B., Vallentin, M., and Sommer, R. Here's my cert, so trust me, maybe?: understanding tls errors on the web. In: WWW '13. 2013.

[15] Alghamdi, K., Alqazzaz, A., Liu, A., and Ming, H. Iotverif: an automated tool to verify ssl/tls certificate validation in android mqtt client applications. In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. CODASPY '18. Association for Computing Machinery, Tempe, AZ, USA, 2018, 95–102.

[16] Allen, C. and Dierks, T. *The TLS Protocol Version 1.0*. https://rfc-editor.org/rfc/rfc2246.txt. Last visited: 09/22/2020. Jan. 1999.

[17] 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*. MSR '16. ACM, Austin, Texas, 2016, 468–471.

[18] Almanee, S., Unal, A., Payer, M., and Garcia, J. *Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code*. 2021. arXiv: 1911.09716 [cs.CR].

[19] Amour, L. S. and Petullo, W. M. Improving application security through TLS-library redesign. In: *Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2015, 75–94.

[20] *An Asynchronous HTTP Library for Android*. https://github.com/android-async-http/android-async-http. Last visited: 09/22/2020.

[21] André, C. *GMail Android App Insecure Network Security Configuration*. https://labs.integrity.pt/articles/Gmail-Android-app-insecure-Network-Security-Configuration/. Last visited: 09/22/2020. 2018.

[22] *Androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)* https://github.com/androguard/androguard. Last visited: 09/22/2020.

[23] *Android 7.0 unable to capture https packets*. https://www.cnblogs.com/0616--ataozhijia/p/9766682.html. Last visited: 09/22/2020.

[24] *Android 8: Cleartext HTTP traffic not permitted*. https://stackoverflow.com/questions/45940861/android-8-cleartext-http-traffic-not-permitted. Last visited: 09/22/2020.

[25] *Android M and the war on cleartext traffic*. https://koz.io/android-m-and-the-war-on-cleartext-traffic/. Last visited: 09/22/2020. 2015.

[26] *Android OS market share of smartphone sales to end users from 2009 to 2019*. https://www.statista.com/statistics/216420/global-market-share-forecast-of-smartphone-operating-systems/. Last visited: 03/30/2021. Mar. 2020.

[27] *Android Root CAs*. https://android.googlesource.com/platform/system/ca-certificates/+/master/files/. Last visited: 09/22/2020.

[28] *Android WebView setCertificate issues SSL problems*. https://stackoverflow.com/questions/6511434/android-webview-setcertificate-issues-ssl-problems/57951506#57951506. Last visited: 09/22/2020.

[29] *Andromo*. http://www.andromo.com/. Last visited: 12/01/2017.

[30] *Apache Cordova*. https://cordova.apache.org/. Last visited: 12/01/2017.

[31] API, A. *Android TLS API*. https://developer.android.com/training/articles/security-ssl.html.

[32] *App Gyver*. https://www.appgyver.io/. Last visited: 12/01/2017.

[33] *App Titan*. http://www.apptitan.de/. Last visited: 12/01/2017.

[34] *App Yourself*. http://appyourself.net/. Last visited: 12/01/2017.

[35] *Appcelerator*. http://www.appcelerator.com/product/. Last visited: 12/01/2017.

[36] *Appconfector*. https://www.appconfector.de/. Last visited: 12/01/2017.

[37] *appery.io*. https://appery.io/. Last visited: 12/01/2017.

[38] *Appindex*. http://appindex.com/blog/app-builders-app-makers-list/. Last visited: 12/01/2017.

[39] *Appinventor*. http://appinventor.mit.edu/explore/. Last visited: 12/01/2017.

[40] *Application Crash Reports for Android*. https://github.com/ACRA/acra. Last visited: 09/22/2020.

[41] *Application-only authentication | Twitter Developer*. https://dev.twitter.com/oauth/application-only. Last visited: 12/01/2017.

[42] *Applicationcraft*. http://www.applicationcraft.com/. Last visited: 12/01/2017.

[43] *Appmachine*. http://www.appmachine.com/. Last visited: 12/01/2017.

[44] *Appmakr*. http://appmakr.com/. Last visited: 12/01/2017.

[45] *AppMk*. http://www.appmk.com/. Last visited: 12/01/2017.

[46] *Apps Bar*. http://www.appsbar.com/. Last visited: 12/01/2017.

[47] *Apps Builder*. http://www.apps-builder.com/. Last visited: 12/01/2017.

[48] *Apps Geyser*. http://www.appsgeyser.com/. Last visited: 12/01/2017.

[49] *Appsme*. http://www.appsme.com/. Last visited: 12/01/2017.

[50] *Appy Pie*. http://www.appypie.com/. Last visited: 12/01/2017.

[51] *AppYet*. http://www.appyet.com. Last visited: 12/01/2017.

[52] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. le, 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)*. ACM, 2014.

[53] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: analyzing the android permission specification. In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[54] Aviram, N., Schinzel, S., Somorovsk, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohne, S., Engels, S., Paar, C., and Shavitt, Y. DROWN: breaking TLS with SSLv2. In: *Proc. 24th USENIX Security Symposium (SEC'15)*. CVE-2016-0800. Aug. 2016.

[55] Ayer, A. *Domain Validation Vulnerability in Symantec Certificate Authority*. `https://www.agwa.name/blog/post/domain_validation_vulnerability_in_symantec_ca`. Last visited: 05/19/2021. Feb. 2016.

[56] *B4X*. `https://www.b4x.com/`. Last visited: 12/01/2017.

[57] 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*. ACM. 2016, 356–367.

[58] Backes, M., Bugiel, S., Derr, E., Gerling, S., and Hammer, C. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, 2016.

[59] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: *Proc. 25th USENIX Security Symposium (SEC'16)*. USENIX Association, 2016.

[60] Backes, M., Bugiel, S., and Gerling, S. Scippa: system-centric ipc provenance on android. In: *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 2014.

[61] *Ballot 187 - Make CAA Checking Mandatory - CAB Forum*. `https://cabforum.org/2017/03/08/ballot-187-make-caa-checking-mandatory/y`. Last visited: 05/19/2021.

[62] *Ballot SC22 - Reduce Certificate Lifetimes (v2) - CAB Forum*. `https://certificate.transparency.dev/community`. Last visited: 05/19/2021.

[63] Barnes, R., Hoffman-Andrews, J., McCarney, D., and Kasten, J. *Automatic Certificate Management Environment (ACME)*. RFC 8555. Mar. 2019.

[64] *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.0*. `https://cabforum.org/wp-content/uploads/Baseline_Requirements_V1.pdf`. Last visited: 05/19/2021.

[65] Bates, A., Pletcher, J., Nichols, T., Hollembaek, B., Tian, D., Butler, K. R., and Alkhelaifi, A. Securing ssl certificate verification through dynamic linking. In: CCS '14. ACM, Scottsdale, Arizona, USA, 2014, 394–405.

[66] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P. Y., and Zinzindohoue, J. K. A messy state of the union: taming the composite state machines of tls. In: *Proc. 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 2015.

[67] *Biznessapps*. `http://www.biznessapps.com/`. Last visited: 12/01/2017.

[68] 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*. `https://rfc-editor.org/rfc/rfc5280.txt`. Last visited: 09/22/2020. May 2008.

[69] Bojjagani, S. and Sastry, V. N. Vaptai: a threat model for vulnerability assessment and penetration testing of android and ios mobile banking apps. In: *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*. 2017, 77–86.

[70] Book, T., Pridgen, A., and Wallach, D. S. Longitudinal analysis of android ad library permissions. In: *MoST'13*. IEEE, 2013.

[71] Book, T. and Wallach, D. S. A case of collusion: a study of the interface between ad libraries and their apps. In: *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. SPSM '13. Association for Computing Machinery, Berlin, Germany, 2013, 79–86.

[72] Brubaker, C. *Changes to Trusted Certificate Authorities in Android Nougat*. `https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html`. Last visited: 09/22/2020. 2016.

[73] Brubaker, C. *Protecting users with TLS by default in Android P*. `https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html`. Last visited: 09/22/2020. Apr. 2018.

[74] Buhov, D., Huber, M., Merzdovnik, G., and Weippl, E. Pin it! improving android network security at runtime. In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 2016, 297–305.

[75] Butler, E. *Firesheep - codebutler*. `https://codebutler.com/2010/10/24/firesheep/`. Last visited: 05/19/2021. Oct. 2010.

[76] Carbunar, B. and Potharaju, R. A longitudinal study of the google app market. In: *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. 2015, 242–249.

[77] *Cert Spotter - Timeline of PKI Security Failures*. `https://sslmate.com/certspotter/failures`. Last visited: 05/19/2021.

[78] *Certificate Lifetimes*. `https://chromium.googlesource.com/chromium/src/+/HEAD/net/docs/certificate_lifetimes.md`. Last visited: 05/19/2021.

[79] *Certificate pinning. · square/okhttp@83090be · GitHub*. https://github.com/square/okhttp/commit/83090befcca69b44c257b96afb519ca66282ca63. Last visited: 05/31/2021.

[80] *CertificatePinner*. https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html. Last visited: 09/22/2020.

[81] *Charles Web Debugging Proxy • HTTP Monitor / HTTP Proxy*. https://www.charlesproxy.com/. Last visited: 09/22/2020.

[82] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-application Communication in Android. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys '11. ACM, Bethesda, Maryland, USA, 2011, 239–252.

[83] Chin, E. and Wagner, D. Bifocals: analyzing webview vulnerabilities in android applications. In: *Proc. Information Security Applications*. Springer-Verlag, 2014.

[84] *Choose a category and tags for your app or game - Play Console Help*. https://support.google.com/googleplay/android-developer/answer/113475?hl=en. Last visited: 12/01/2017.

[85] 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*. Ed. by Kiayias, A. Springer International Publishing, Cham, 2017, 579–597.

[86] *Chromium Blog: A safer default for navigation: HTTPS*. https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html. Last visited: 05/19/2021. Mar. 2021.

[87] *Chromium Blog: New Chromium security features, June 2011*. https://blog.chromium.org/2011/06/new-chromium-security-features-june.html. Last visited: 05/19/2021. June 2011.

[88] *Chromium Blog: No More Mixed Messages About HTTPS*. https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html. Last visited: 05/19/2021. Oct. 2019.

[89] Clark, J. and van Oorschot, P. C. Sok: ssl and https: revisiting past challenges and evaluating certificate trust model enhancements. In: *2013 IEEE Symposium on Security and Privacy*. 2013, 511–525.

[90] *Community : Certificate Transparency*. https://certificate.transparency.dev/community. Last visited: 05/19/2021.

[91] *Como*. http://www.como.com/. Last visited: 12/01/2017.

[92] Comodo. *Comodo Report of Fraud Incident*. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html. Last visited: 12/01/2017. Mar. 2011.

[93] *Comodo Certificate Issue - Follow Up - Mozilla Security Blog*. https://blog.mozilla.org/security/2011/03/25/comodo-certificate-issue-follow-up/. Last visited: 05/19/2021. 2011.

[94] 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, 65–81.

[95] *Crosswalk*. https://crosswalk-project.org/. Last visited: 12/01/2017.

[96] *CrowdCompass*. http://www.crowdcompass.com/. Last visited: 12/01/2017.

[97] Cunningham, E. *Improving app security and performance on Google Play for years to come*. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html. Last visited: 09/22/2020. Dec. 2017.

[98] *CVE-2016-2107: Padding-oracle attack against AES CBC*. Jan. 2016.

[99] *CVE-2016-2402*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2402. Last visited: 09/22/2020. Feb. 2016.

[100] *Debug your app | Android Developers*. https://developer.android.com/studio/debug. Last visited: 09/22/2020.

[101] Dierks, T. and Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176. Internet Engineering Task Force, Aug. 2008.

[102] Duong, T. and Rizzo, J. *Here come the ⊕ ninjas*. May 2011.

[103] Durumeric, Z., Kasten, J., Bailey, M., and Halderman, J. A. Analysis of the HTTPS Certificate Ecosystem. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. ACM, Barcelona, Spain, 2013, 291–304.

[104] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-wide scanning and its security applications. In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, Washington, D.C., USA, Aug. 2013, 605–620.

[105] *Eachscape*. http://eachscape.com/. Last visited: 12/01/2017.

[106] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. ACM, Berlin, Germany, 2013, 73–84.

[107] 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. In: *Proc. 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[108] Enck, W., Octeau, D., McDaniel, P. D., and Chaudhuri, S. A Study of Android Application Security. In: *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[109] Enck, W., Ongtang, M., and McDaniel, P. Understanding Android Security. In: *Proceedings of the IEEE International Conference on Security & Privacy*. 2009, 50–57.

[110] Evans, C., Palmer, C., and Sleevi, R. *Public Key Pinning Extension for HTTP*. `https://rfc-editor.org/rfc/rfc7469.txt`. Last visited: 09/22/2020. Apr. 2015.

[111] *Facebook | Secure browsing by default*. `https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920/`. Last visited: 05/19/2021.

[112] Fahl, S., Acar, Y., Perl, H., and Smith, M. Why eve and mallory (also) love webmasters: a study on the root causes of ssl misconfigurations. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. ACM, Kyoto, Japan, 2014, 507–512.

[113] Fahl, S., Dechand, S., Perl, H., Fischer, F., Smrcek, J., and Smith, M. Hey, nsa: stay away from my market! future proofing app markets against powerful attackers. In: CCS '14. ACM, Scottsdale, Arizona, USA, 2014, 1143–1155.

[114] 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: *Proceedings of the 2012 ACM conference on Computer and communications security*. Proc. 19th ACM Conference on Computer and Communication Security (CCS'12). ACM, Raleigh, North Carolina, USA, 2012, 50–61.

[115] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. In: CCS '13. ACM, Berlin, Germany, 2013, 49–60.

[116] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In: *Proc. 18th ACM Conference on Computer and Communication Security (CCS'11)*. ACM, 2011.

[117] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: user attention, comprehension, and behavior. In: *Proc. 8th Symposium on Usable Privacy and Security (SOUPS '12)*. ACM, 2012.

[118] Felt, A. P., Ainslie, A., Reeder, R. W., Consolvo, S., Thyagaraja, S., Bettes, A., Harris, H., and Grimes, J. Improving ssl warnings: comprehension and adherence. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. ACM, Seoul, Republic of Korea, 2015, 2893–2902.

[119] Felt, A. P., Barnes, R., King, A., Palmer, C., Bentzel, C., and Tabriz, P. Measuring HTTPS adoption on the web. In: *Proc. 26th Usenix Security Symposium (SEC'17)*. USENIX Association, Vancouver, BC, Aug. 2017, 1323–1338.

[120] Felt, A. P., Reeder, R. W., Ainslie, A., Harris, H., Walker, M., Thompson, C., Acer, M. E., Morant, E., and Consolvo, S. Rethinking connection security indicators. In: *Symposium on Usable Privacy and Security*. 2016.

[121] *Firefox 83 introduces HTTPS-Only Mode - Mozilla Security Blog*. `https://blog.mozilla.org/security/2020/11/17/firefox-83-introduces-https-only-mode/`. Last visited: 05/19/2021. Nov. 2020.

[122] 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)*. IEEE, 2017.

[123] Fisher, A. *How Companies are developing more apps with fewer developers*. `http://fortune.com/2016/08/30/quickbase-coding-apps-developers/`. Last visited: 11/29/2017. Aug. 2016.

[124] *GameSalad*. `http://gamesalad.com/`. Last visited: 12/01/2017.

[125] Gartner. *Gartner Says Citizen Developers Will Build at Least 25 Percent of New Business Applications by 2014*. `http://www.gartner.com/newsroom/id/1744514`. Last visited: 11/29/2017. July 2011.

[126] 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: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, Raleigh, North Carolina, USA, 38–49.

[127] Georgiev, M., Jana, S., and Shmatikov, V. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks. In: *2014 Network and Distributed System Security (NDSS '14)*. San Diego, Feb. 2014.

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

[129] Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., and Choi, H. Adrob: examining the landscape and impact of android application plagiarism. In: *Proc. 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*. ACM, 2013.

[130] Glanz, L., Amann, S., Eichberg, M., Reif, M., Hermann, B., Lerch, J., and Mezini, M. Codematch: obfuscation won't conceal your repackaged app. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Association for Computing Machinery, Paderborn, Germany, 2017, 638–648.

[131] Gonzalez, H., Stakhanova, N., and Ghorbani, A. Measuring code reuse in android apps. In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. 2016, 187–195.

[132] *Good Barber*. `http://www.goodbarber.com/`. Last visited: 12/01/2017.

[133] *Goodbye launcher drawables, hello mipmaps! | Android By Code*. `https://androidbycode.wordpress.com/2015/02/14/goodbye-launcher-drawables-hello-mipmaps/`. Last visited: 12/01/2017.

[134] Google. *Android Developers Blog: Using Cryptography to Store Credentials Safely*. `http://android-developers.blogspot.de/2013/02/using-cryptography-to-store-credentials.html`. Last visited: 11/29/2017. 2013.

[135] Google. *How to address WebView SSL Error Handler alerts in your apps.* https://support.google.com/faqs/answer/7071387. Last visited: 09/22/2020. 2016.

[136] Google. *How to fix apps containing an unsafe implementation of TrustManager.* https://support.google.com/faqs/answer/6346016. Last visited: 09/22/2020. 2016.

[137] Google. *How to resolve Insecure HostnameVerifier.* https://support.google.com/faqs/answer/7188426. Last visited: 09/22/2020. 2017.

[138] Google. *Android 6.0 Changes.* https://developer.android.com/about/versions/marshmallow/android-6.0-changes. Last visited: 09/22/2020.

[139] Google. *Android 7.0 for Developers.* https://developer.android.com/about/versions/nougat/android-7.0. Last visited: 09/22/2020.

[140] Google. *Android 8.0 for Developers.* https://developer.android.com/about/versions/oreo/android-8.0-changes. Last visited: 09/22/2020.

[141] Google. *Android Developers: Security Tips.* https://developer.android.com/training/articles/security-tips.html. Last visited: 11/29/2017.

[142] Google. *App security improvement program.* https://developer.android.com/google/play/asi. Last visited: 09/22/2020.

[143] Google. *Behavior changes: apps targeting API level 28+.* https://developer.android.com/about/versions/pie/android-9.0-changes-28#framework-security-changes. Last visited: 09/22/2020.

[144] Google. *Security Enhancements in Android 6.0.* https://source.android.com/security/enhancements/enhancements60. Last visited: 09/22/2020.

[145] Google. *Upload App.* https://support.google.com/googleplay/android-developer/answer/113469#targetsdk. Last visited: 09/22/2020.

[146] Google. *URL | Android Developers.* https://developer.android.com/reference/java/net/URL. Last visited: 09/22/2020.

[147] *Google Online Security Blog: Enhancing digital certificate security.* https://security.googleblog.com/2013/01/enhancing-digital-certificate-security.html. Last visited: 05/19/2021. Jan. 2013.

[148] *Google Play.* https://play.google.com/. Last visited: 03/30/2021.

[149] *Google Play Blocker: Unsafe SSL TrustManager Defined #1260.* https://github.com/android-async-http/android-async-http/issues/1260. Last visited: 09/22/2020.

[150] *Google play python API.* https://github.com/NoMore201/googleplay-api. Last visited: 09/22/2020.

[151] *Google Safe Browsing | Google Developers.* https://developers.google.com/safe-browsing/. Last visited: 12/01/2017.

[152] Gorski, P. L., Iacono, L. L., Wermke, D., Stransky, C., Möller, S., Acar, Y., and Fahl, S. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In: *Proc. 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX Association, 2018.

[153] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock Android smartphones. In: *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[154] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: scalable and accurate zero-day android malware detection. In: *Proc. 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. ACM, 2012.

[155] Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In: *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WISEC '12. Association for Computing Machinery, Tucson, Arizona, USA, 2012, 101–112.

[156] Green, M. and Smith, M. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy* 14, 5 (2016), 40–46.

[157] Greenwald, G. *NSA Prism program taps in to user data of Apple, Google and others | US national security | The Guardian*. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data. Last visited: 05/19/2021. June 2013.

[158] Gross, D. *Apple trademarks 'There's an app for that' - CNN.com*. http://edition.cnn.com/2010/TECH/mobile/10/12/app.for.that/index.html. Last visited: 03/30/2021. Oct. 2010.

[159] Hallam-Baker, P. and Stradling, R. *DNS Certification Authority Authorization (CAA) Resource Record*. RFC 6844. Jan. 2013.

[160] 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, 519–534.

[161] Helme, S. *Alexa Top 1 Million Crawl - August 2016*. https://scotthelme.co.uk/alexa-top-1-million-crawl-aug-2016/. Last visited: 05/19/2021. July 2016.

[162] Helme, S. *I'm giving up on HPKP*. https://scotthelme.co.uk/im-giving-up-on-hpkp/. Last visited: 05/19/2021. Aug. 2017.

[163] Helme, S. *HPKP is no more!* https://scotthelme.co.uk/hpkp-is-no-more/. Last visited: 05/19/2021. Jan. 2020.

[164] Hinchcliffe, D. *The advent of the citizen developer*. http://www.zdnet.com/article/the-advent-of-the-citizen-developer/. Last visited: 11/29/2017. Apr. 2016.

[165] Hodges, J., Jackson, C., and Barth, A. *RFC 6797 HTTP Strict Transport Security (HSTS)*. https://datatracker.ietf.org/doc/rfc6797/?include_text=1. Nov. 2012.

[166] Hoffman, P. and Schlyter, J. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698 (Proposed Standard). http://tools.ietf.org/html/rfc6698. IETF, 2012.

[167] Hongmu, H., Li, R., and Tang, J. Identify and Inspect Libraries in Android Applications. *Wireless Personal Communications* 103 (Nov. 2018), 1–13.

[168] *HostnameVerifier*. https://developer.android.com/reference/kotlin/javax/net/ssl/HostnameVerifier. Last visited: 09/22/2020.

[169] *HostnameVerifierDetector.java - platform/tools/base - Git at Google*. https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks/BadHostnameVerifierDetector.java. Last visited: 03/30/2021.

[170] *HTTPS encryption on the web – Google Transparency Report*. https://transparencyreport.google.com/https/overview. Last visited: 05/19/2021.

[171] *HTTPS Everywhere | Electronic Frontier Foundation*. https://www.eff.org/https-everywhere. Last visited: 05/19/2021.

[172] *Incident Report - OCR*. https://www.mail-archive.com/dev-security-policy@lists.mozilla.org/msg04654.html. Last visited: 05/19/2021. 2016.

[173] *Incidents involving the CA WoSign*. https://www.mail-archive.com/dev-security-policy@lists.mozilla.org/msg03665.html. Last visited: 05/19/2021. 2016.

[174] *Intent To Deprecate And Remove: Public Key Pinning*. https://groups.google.com/a/chromium.org/g/blink-dev/c/he9tr7p3rZ8/m/eNMwKPmUBAAJ. Last visited: 05/19/2021. Oct. 2017.

[175] *ionic*. http://ionicframework.com/. Last visited: 12/01/2017.

[176] *Java android - uplaud apk and google play security alert*. https://stackoverflow.com/questions/43847629/java-android-uplaud-apk-and-google-play-security-alert. Last visited: 09/22/2020.

[177] *Java android . Google play security alert for insecure TrustManager*. https://stackoverflow.com/questions/43777599/java-android-google-play-security-alert-for-insecure-trustmanager. Last visited: 09/22/2020.

[178] Jin, X., Hu, X., Ying, K., Du, W., Yin, H., and Peri, G. N. Code injection attacks on html5-based mobile apps: characterization, detection and mitigation. In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[179] *JSoup Issue: TLS Certificate Bypassable, throws warnings #912*. https://github.com/jhy/jsoup/issues/912. Last visited: 09/22/2020.

[180] *jsoup: Java HTML Parser, with best of DOM, CSS, and jquery*. https://github.com/jhy/jsoup/. Last visited: 09/22/2020.

[181] Kafle, K., Moran, K., Manandhar, S., Nadkarni, A., and Poshyvanyk, D. A study of data store-based home automation. In: CODASPY'19. Association for Computing Machinery, Richardson, Texas, USA, 2019, 73–84.

[182] Kaufman, J. *History of HTTPS Usage*. https://www.jefftk.com/p/history-of-https-usage. Last visited: 05/31/2021.

[183] Kotzias, P., Razaghpanah, A., Amann, J., Paterson, K. G., Vallina-Rodriguez, N., and Caballero, J. Coming of age: a longitudinal study of tls deployment. In: *Proceedings of the Internet Measurement Conference 2018*. IMC '18. Association for Computing Machinery, Boston, MA, USA, 2018, 415–428.

[184] Kozyrakis, J. *An examination of ineffective certificate pinning implementations*. https://www.synopsys.com/blogs/software-security/ineffective-certificate-pinning-implementations/. Last visited: 09/22/2020. 2016.

[185] Kranch, M. and Bonneau, J. Upgrading HTTPS in mid-air: an empirical study of strict transport security and key pinning. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[186] Krombholz, K., Mayer, W., Schmiedecker, M., and Weippl, E. "i have no idea what i'm doing" - on the usability of deploying HTTPS. In: *Proc. 26th Usenix Security Symposium (SEC'17)*. USENIX Association, Vancouver, BC, 2017, 1339–1356.

[187] 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*. ASE'17. IEEE Press, Urbana-Champaign, IL, USA, 2017, 931–936.

[188] Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., and Kapitza, R. Attack surface metrics and automated compile-time OS kernel tailoring. In: *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[189] Langley, A. *ImperialViolet - Revocation checking and Chrome's CRL*. https://www.imperialviolet.org/2012/02/05/crlsets.html. Last visited: 05/19/2021. Feb. 2012.

[190] Langley, A. *Google Online Security Blog: Enhancing digital certificate security*. https://security.googleblog.com/2014/07/maintaining-digital-certificate-security.html. Last visited: 05/19/2021. July 2014.

[191] Langley, A. *Google Online Security Blog: Enhancing digital certificate security*. https://security.googleblog.com/2015/03/maintaining-digital-certificate-security.html. Last visited: 05/19/2021. Mar. 2015.

[192] Laurie, B., Langley, A., and Kasper, E. *RFC 6962 Certificate Transparency*. http://tools.ietf.org/html/rfc6962. June 2013.

[193] *Leaving Beta, New Sponsors - Let's Encrypt.* https://letsencrypt.org/2016/04/12/leaving-beta-new-sponsors.html. Last visited: 05/19/2021. Apr. 2016.

[194] *Let's Encrypt on Twitter.* https://mobile.twitter.com/letsencrypt/status/1413832381233049601. Last visited: 07/12/2021.

[195] *Let's Encrypt Stats - Let's Encrypt.* https://letsencrypt.org/stats/. Last visited: 05/19/2021.

[196] *Let's Encrypt: Delivering SSL/TLS Everywhere - Let's Encrypt.* https://letsencrypt.org/2014/11/18/announcing-lets-encrypt.html. Last visited: 05/19/2021. Nov. 2014.

[197] Li, L., Bissyandé, T., Klein, J., and Le Traon, Y. An investigation into the use of common libraries in android apps. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* Vol. 1. 2016, 403–414.

[198] Li, L., Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Traon, Y. L. Mining families of android applications for extractive spl adoption. In: *Proceedings of the 20th International Systems and Software Product Line Conference.* SPLC '16. Association for Computing Machinery, Beijing, China, 2016, 271–275.

[199] Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., and Huo, W. Libd: scalable and precise third-party library detection in android markets. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017, 335–346.

[200] Linares-Vásquez, M., Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D. Revisiting android reuse studies in the context of code obfuscation and library usages. In: *Proceedings of the 11th Working Conference on Mining Software Repositories.* MSR 2014. Association for Computing Machinery, Hyderabad, India, 2014, 242–251.

[201] Liu, B., Liu, B., Jin, H., and Govindan, R. Efficient privilege de-escalation for ad libraries in mobile apps. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* MobiSys '15. Association for Computing Machinery, Florence, Italy, 2015, 89–103.

[202] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In: CCS '12. ACM, Raleigh, North Carolina, USA, 2012, 229–240.

[203] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on WebView in the Android system. In: *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11).* ACM, 2011.

[204] Ma, Z., Wang, H., Guo, Y., and Chen, X. Libradar: fast and accurate detection of third-party libraries in android apps. In: *Proceedings of the 38th International Conference on Software Engineering Companion.* ICSE '16. Association for Computing Machinery, Austin, Texas, 2016, 653–656.

[205] *mallodroid: Find broken SSL certificate validation in Android Apps.* https://github.com/sfahl/mallodroid. Last visited: 09/22/2020.

[206] Marlinspike, M. New Tricks for Defeating SSL in Practice. In: *BlackHat Europe 2009*. 2009.

[207] Marlinspike, M. *SSL And The Future Of Authenticity*. In BlackHat USA. 2011.

[208] Marlinspike, M. *TACK: Trust Assertion for Certificate Keys*. http://tack.io/. Last visited: 07/14/2021. 2013.

[209] Marlinspike, M. *Android Pinning*. https://github.com/moxie0/AndroidPinning. Last visited: 07/14/2021.

[210] *MediaDevices.getUserMedia() - Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia. Last visited: 05/19/2021.

[211] *Meet Android Studio | Android Developers*. http://developer.android.com/tools/studio/index.html. Last visited: 12/01/2017.

[212] *Mippin App Factory*. http://www.mippin.com/appfactory/. Last visited: 12/01/2017.

[213] *Misissued/Suspicious Symantec Certificates*. https://www.mail-archive.com/dev-security-policy@lists.mozilla.org/msg05455.html. Last visited: 05/19/2021. 2017.

[214] *Mobile Roadie*. http://mobileroadie.com/. Last visited: 12/01/2017.

[215] *Mobincube*. http://www.mobincube.com/. Last visited: 12/01/2017.

[216] Möller, B., Duong, T., and Kotowicz, K. *This POODLE Bites: Exploiting the SSL 3.0 Fallback (Security Advisory)*. https://www.openssl.org/~bodo/ssl-poodle.pdf. Last visited: 12/01/2017. Sept. 2014.

[217] Mutchler, P., Doupé, A., Mitchell, J., Kruegel, C., and Vigna, G. A Large-Scale Study of Mobile Web App Security. In: *Proc. 2015 Mobile Security Technologies Workshop (MoST'15)*. IEEE, 2015.

[218] Narayanan, A., Chen, L., and Chan, C. Addetect: automated detection of android ad libraries using semantic analysis. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 2014, 1–6.

[219] *NativeScript*. https://github.com/NativeScript/. Last visited: 12/01/2017.

[220] *Network security configuration*. https://developer.android.com/training/articles/security-config. Last visited: 09/22/2020.

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

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

[223] *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. Last visited: 09/22/2020.

[224] 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)*. ACM, 2017.

[225] Nightingale, J. *DigiNotar Removal Follow Up*. https://blog.mozilla.org/security/2011/09/02/diginotar-removal-follow-up/. Last visited: 12/01/2017. Sept. 2011.

[226] 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.

[227] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Traon, Y. L. Effective inter-component communication mapping in android: an essential step towards holistic security analysis. In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[228] Oltrogge, M. and Fahl, S. *Android Pinning - Frontend and Backend*. https://zenodo.org/record/5091357. Version 1.0. Last visited: 07/21/2021. Aug. 2015.

[229] 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)*. ACM. 2015, 1–6.

[230] *Open Handset Alliance™*. https://www.openhandsetalliance.com/. Last visited: 03/30/2021.

[231] OWASP. *OWASP Certificate Pinning Guide*. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning.

[232] *Paradise Apps*. http://www.paradiseapps.net/. Last visited: 12/01/2017.

[233] Pearce, P., Porter Felt, A., Nunez, G., and Wagner, D. AdDroid: privilege separation for applications and advertisers in Android. In: *Proc. 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS'12)*. ACM, 2012.

[234] Perl, H., Fahl, S., and Smith, M. You won't be needing these any more: on removing unused certificates from trust stores. In: *Proceedings of 18th International Conference on Financial Cryptography and Data Security (FC 2014)*. Springer Berlin Heidelberg, 2014.

[235] *PhoneGap SSL Certificate Checker plugin*. https://github.com/EddyVerbruggen/SSLCertificateChecker-PhoneGap-Plugin. Last visited: 09/22/2020.

[236] *Platform Architecture | Android Developers*. https://developer.android.com/guide/platform. Last visited: 03/30/2021.

[237] *PMD Source Code Analyzer*. https://pmd.github.io/. Last visited: 03/30/2021.

[238] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: *Proceedings of the 21st Network and Distributed Systems Security Symposium*. NDSS. San Diego, CA, Feb. 2014.

[239] Possemato, A. and Fratantonio, Y. Towards HTTPS everywhere on android: we are not there yet. In: *Proc. 29th Usenix Security Symposium (SEC'20)*. USENIX Association, Aug. 2020, 343–360.

[240] *Post-PRISM, Google Confirms Quietly Moving To Make All Searches Secure, Except For Ad Clicks*. https://searchengineland.com/post-prism-google-secure-searches-172487. Last visited: 05/19/2021.

[241] Qualys. *SSL Labs*. https://www.ssllabs.com/ssl-pulse/. Last visited: 05/31/2021. 2021.

[242] *Quickmobile*. http://www.quickmobile.com/. Last visited: 12/01/2017.

[243] Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. (Cryptoguard: high precision detection of cryptographic vulnerabilities in massive-sized java projects. In: CCS'19. Association for Computing Machinery, London, United Kingdom, 2019, 2455–2472.

[244] 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*. CoNEXT'17. ACM, Incheon, Republic of Korea, 2017, 350–362.

[245] *React Native*. https://github.com/facebook/react-native. Last visited: 12/01/2017.

[246] *Reducing TLS Certificate Lifespans to 398 Days - Mozilla Security Blog*. https://blog.mozilla.org/security/2020/07/09/reducing-tls-certificate-lifespans-to-398-days/. Last visited: 05/19/2021. July 2020.

[247] *Remove HTTP-Based Public Key Pinning - Chrome Platform Status*. https://www.chromestatus.com/feature/5903385005916160. Last visited: 05/19/2021. 2019.

[248] Rescorla, E. *RFC 2818 HTTP Over TLS*. http://tools.ietf.org/html/rfc2818. May 2000.

[249] Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. https://rfc-editor.org/rfc/rfc8446.txt. Last visited: 09/22/2020. Aug. 2018.

[250] *Rho Mobile Suite*. http://rhomobile.com/. Last visited: 12/01/2017.

[251] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C. *RFC 6960 X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. https://tools.ietf.org/html/rfc6960. June 2013.

[252] Scheitle, Q., Chung, T., Hiller, J., Gasser, O., Naab, J., Rijswijk-Deij, R. van, Hohlfeld, O., Holz, R., Choffnes, D., Mislove, A., and Carle, G. A first look at certification authority authorization (caa). *SIGCOMM Comput. Commun. Rev.* 48, 2 (May 2018), 10–23.

[253] Schrauger, S. *The story of how WoSign gave me an SSL certificate for GitHub.com | Schrauger.com*. https://www.schrauger.com/the-story-of-how-wosign-gave-me-an-ssl-certificate-for-github-com. Last visited: 05/19/2021. Aug. 2016.

[254] *Seattle Cloud*. http://seattleclouds.com/. Last visited: 12/01/2017.

[255] Seattle Cloud. *Login Page Type Tutorial*. http://seattleclouds.com/login-page-type-tutorial. Last visited: 11/29/2017. 2016.

[256] *Secure contexts - Web security | MDN*. https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts. Last visited: 05/19/2021.

[257] Shin, D. and Lopes, R. An empirical study of visual security cues to prevent the sslstripping attack. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, 287–296.

[258] *Should Your Business Build a Mobile App? - businessnewsdaily.com*. http://www.businessnewsdaily.com/4901-best-app-makers-creators.html. Last visited: 12/01/2017.

[259] *Shoutem*. http://www.shoutem.com/. Last visited: 12/01/2017.

[260] Sleevi, R. *Google Online Security Blog: Sustaining Digital Certificate Security*. https://security.googleblog.com/2015/10/sustaining-digital-certificate-security.html. Last visited: 05/19/2021. Oct. 2015.

[261] Soh, C., Kuan Tan, H., Arnatovich, Y., Narayanan, A., and Wang, L. Libsift: automated detection of third-party libraries in android applications. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 2016, 41–48.

[262] Son, S., Daehyeok, G., Kaist, K., and Shmatikov, V. What mobile ads know about mobile users. In: *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*. 2015.

[263] 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. In: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[264] *SSL Ratios (public) - Mozilla Data Documentation*. https://docs.telemetry.mozilla.org/datasets/other/ssl/reference.html. Last visited: 05/19/2021.

[265] *SSL Server Test (Powered by Qualys SSL Labs)*. https://www.ssllabs.com/ssltest/. Last visited: 12/01/2017.

[266] *SSL/TLS and PKI History.* https://www.feistyduck.com/ssl-tls-and-pki-history/. Last visited: 05/19/2021.

[267] Stark, E. *Rolling out Public Key Pinning with HPKP Reporting | Web.* https://developers.google.com/web/updates/2015/09/HPKP-reporting-with-chrome-46. Last visited: 05/19/2021. Aug. 2015.

[268] *StartEncrypt considered harmful today - Computest.* https://www.computest.nl/nl/knowledge-platform/blog/startencrypt-considered-harmful-today/. Last visited: 05/19/2021. June 2016.

[269] Stengel, K., Schmaus, F., and Kapitza, R. Esseos: haskell-based tailored services for the cloud. In: *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware.* ARM '13. ACM, Beijing, China, 2013.

[270] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating user privacy in android ad libraries. In: *Proc. 2012 Mobile Security Technologies Workshop (MoST'12).* IEEE, 2012.

[271] *Stop Certificate Pinning | DigiCert.com.* https://www.digicert.com/blog/certificate-pinning-what-is-certificate-pinning. Last visited: 05/19/2021.

[272] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and Cranor, L. F. Crying wolf: an empirical study of ssl warning effectiveness. In: SSYM'09. 2009.

[273] *SwipeCardView network_security_config.xml.* https://github.com/Gxyong/SwipeCardView/blob/master/app/src/main/res/xml/network_security_config.xml. Last visited: 09/22/2020.

[274] 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).* IEEE, 2014.

[275] *The Titanium SDK and Certificate Validation.* https://devblog.axway.com/mobile-apps/the-titanium-sdk-and-certificate-validation/. Last visited: 05/31/2021. 2012.

[276] *Timeline of HTTPS adoption.* https://timelines.issarice.com/wiki/Timeline_of_HTTPS_adoption#Full_timeline. Last visited: 05/31/2021. 2021.

[277] Tiwari, A., Prakash, J., Groß, S., and Hammer, C. Ludroid: a large scale analysis of android – web hybridization. In: *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM).* 2019, 256–267.

[278] *Tobit Chayns.* http://en.tobit.com/chayns. Last visited: 12/01/2017.

[279] Troncoso, C., Payer, M., Hubaux, J.-P., Salathé, M., Larus, J., Bugnion, E., Lueks, W., Stadler, T., Pyrgelis, A., Antonioli, D., Barman, L., Chatel, S., Paterson, K., Čapkun, S., Basin, D., Beutel, J., Jackson, D., Roeschlin, M., Leu, P., Preneel, B., Smart, N., Abidin, A., Gürses, S., Veale, M., Cremers, C., Backes, M., Tippenhauer, N. O., Binns, R., Cattuto, C., Barrat, A., Fiore, D., Barbosa, M., Oliveira, R., and Pereira, J. *Decentralized Privacy-Preserving Proximity Tracing.* 2020. arXiv: 2005.12273 [cs.CR].

[280] *TrustAllX509TrustManager issue #909.* https://github.com/jhy/jsoup/issues/909. Last visited: 09/22/2020.

[281] *TrustAllX509TrustManagerDetector.java - platform/tools/base - Git at Google.* https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks/TrustAllX509TrustManagerDetector.java. Last visited: 03/30/2021.

[282] *TrustKit-Android: Easy SSL pinning validation and reporting for Android.* https://github.com/datatheorem/TrustKit-Android. Last visited: 09/22/2020.

[283] *TrustKit-Android: Sample NSC file.* https://github.com/datatheorem/TrustKit-Android/blob/master/app/src/main/res/xml/network_security_config.xml. Last visited: 09/22/2020.

[284] *Trustwave admits issuing man-in-the-middle digital certificate; Mozilla debates punishment | Computerworld.* https://www.computerworld.com/article/2501291/trustwave-admits-issuing-man-in-the-middle-digital-certificate--mozilla-debates-punishment.html. Last visited: 05/19/2021. Feb. 2012.

[285] *Unsafe implementation of X509TrustManager #374.* https://github.com/ACRA/acra/issues/374. Last visited: 09/22/2020.

[286] *Upload failed You uploaded a debuggable APK.* https://github.com/phonegap/build/issues/436. Last visited: 09/22/2020.

[287] *Use X509TrustManager for SSL in android.* https://stackoverflow.com/questions/49650900/use-x509trustmanager-for-ssl-in-android. Last visited: 09/22/2020.

[288] *Using Service Workers - Web APIs | MDN.* https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers. Last visited: 05/19/2021.

[289] Vallina-Rodriguez, N., Amann, J., Kreibich, C., Weaver, N., and Paxson, V. A tangled mass: the android root certificate stores. In: CoNEXT '14. ACM, Sydney, Australia, 2014, 141–148.

[290] Vasco.com. *DigiNotar reports security incident.* Last visited: 12/01/2017. 2011.

[291] Viennot, N., Garcia, E., and Nieh, J. A measurement study of google play. In: *Proc. 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14).* ACM, 2014.

[292] Wang, H., Guo, Y., Ma, Z., and Chen, X. Wukong: a scalable and accurate two-phase approach to android app clone detection. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* ISSTA 2015. Association for Computing Machinery, Baltimore, MD, USA, 2015, 71–82.

[293] Wang, R., Xing, L., Wang, X., and Chen, S. Unauthorized origin crossing on mobile platforms: threats and mitigation. In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13).* ACM, 2013.

[294] Wang, Y., Wu, H., Zhang, H., and Rountev, A. Orlis: obfuscation-resilient library detection for android. In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '18. Association for Computing Machinery, Gothenburg, Sweden, 2018, 13–23.

[295] Wang, Y., Liu, X., Mao, W., and Wang, W. Dcdroid: automated detection of ssl/tls certificate verification vulnerabilities in android apps. In: *Proceedings of the ACM Turing Celebration Conference - China*. ACM TURC '19. Association for Computing Machinery, Chengdu, China, 2019.

[296] *WebView API for SSL Pinning [37004921] - Visible to Public - Issue Tracker*. `https://issuetracker.google.com/issues/37004921`. Last visited: 03/30/2021.

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

[298] Wei, X., Wolf, M., Guo, L., Lee, K., Huang, M., and Niu, N. Emphassl: towards emphasis as a mechanism to harden networking security in android apps. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016, 1–6.

[299] 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: *Proc. 29th Usenix Security Symposium (SEC'20)*. USENIX Association, Aug. 2020, 289–305.

[300] Weiser, M. Program Slicing. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* VOL. SE-10, NO. 4 (1984), 352–357.

[301] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: improving ssh-style host authentication with multi-path probing. In: ATC'08. USENIX Association, Boston, Massachusetts, 2008, 321–334.

[302] 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)*. ACM, 2018.

[303] *What are the best mobile app creators for non-coders, both free and paid? - Quora*. `https://www.quora.com/What-are-the-best-mobile-app-creators-for-non-coders-both-free-and-paid`. Last visited: 12/01/2017.

[304] *What's the Best Mobile App Builder? | We Rock Your Web*. `http://www.werockyourweb.com/mobile-app-builder/`. Last visited: 12/01/2017.

[305] *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/`. Last visited: 09/22/2020.

[306] *X509TrustManager*. `https://developer.android.com/reference/javax/net/ssl/X509TrustManager`. Last visited: 09/22/2020.

[307] *Xamarin*. `https://xamarin.com/`. Last visited: 12/01/2017.

[308] Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., and Liu, Y. Automated third-party library detection for android applications: are we there yet? In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* ASE '20. Association for Computing Machinery, Virtual Event, Australia, 2020, 919–930.

[309] Zhang, J., Beresford, A. R., and Kollmann, S. A. Libid: reliable identification of obfuscated third-party android libraries. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2019. Association for Computing Machinery, Beijing, China, 2019, 55–65.

[310] Zhang, Y., Dai, J., Zhang, X., Huang, S., Yang, Z., Yang, M., and Chen, H. Detecting third-party libraries in android applications with high precision and recall. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 2018, 141–152.

[311] Zhou, Y. and Jiang, X. Detecting passive content leaks and pollution in Android applications. In: *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13).* The Internet Society, 2013.

[312] *ZONER AntiVirus in English.* `https://zonerantivirus.com/`. Last visited: 03/30/2021.

[313] Zuo, C., Wu, J., and Guo, S. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* ASIA CCS '15. Association for Computing Machinery, Singapore, Republic of Singapore, 2015, 591–596.

[314] Zusman, M. *DEFCON 17: Criminal Charges are not pursued: Hacking PKI.* `https://defcon.org/images/defcon-17/dc-17-presentations/defcon-17-zusman-hacking_pki.pdf`. Last visited: 05/19/2021. 2018.

A

App Generators

Table A.1: Classification and fingerprinting of SAF (cf. 5.2.1) and Developer-as-a-Service (DaaS) (cf. 5.2.3) application generators sorted by category and app count. One single fingerprinting feature suffices to uniquely classify an application generator.

	Classification					Fingerprinting Features						
	User-written Code	Freeware	Multi-platform	Components	Publishing	Package Name	Code Namespace	Files in Package	File Content	Sign. Cert.	Sign. Cert. Subject	Market (# of apps)
StandAlone Frameworks												
Xamarin [307]	●	●	○	○	○	○	●	●	○	○	○	206,433
Apache Cordova [30]	●	●	●	○	○	○	●	○	○	○	○	143,464
Crosswalk [95]	●	●	●	○	○	○	●	○	○	○	○	18,772
AppMk [45]	○	○	○	●	○	○	●	○	○	○	●	15,251
ionic [175]	●	●	●	○	○	○	●	○	○	○	○	14,564
Appcelerator [35]	●	○	●	○	○	○	●	○	○	●	●	14,227
b4x [56]	●	○	○	○	○	○	●	○	○	○	○	9,494
Game Salad [124]	○	○	○	●	○	●	●	●	○	○	○	5,677
React Native [245]	●	●	●	○	○	○	●	○	○	○	○	3,589
Phonegap [30]	●	●	●	○	○	●	●	○	○	○	○	3,297
Native Script [219]	●	○	●	○	○	○	●	○	○	○	○	211
Developer-as-a-Service												
CrowdCompass [96]	V	V	○	○	V	●	●	○	○	○	○	2,726
Quickmobile [242]	V	○	●	●	V	●	●	○	●	●	●	624

● = yes/applies; ; ◐ = applies partly; ○ = no/does not apply; V = varies

B

Network Security Configuration

Table B.1: 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 B.2: 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 B.3: 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 B.4: 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

Table B.5: 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 B.6: 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 B.1: Empty TrustManager - Accepts all certificates

```
1 @Override
2 public void checkServerTrusted(X509Certificate[] chain, String /
      authType) throws CertificateException {
3 }
```

Listing B.2: Empty HostnameVerifier - Accepts all hostnames

```
1 @Override
2 public boolean verify(String host, SSLSession session) {
3     return true;
4 }
```

Listing B.3: NSC permitting HTTP traffic again

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <network-security-config>
3 ...
4 <base-config cleartextTrafficPermitted="true">
5 ...
6 </base-config>
7 ...
8 </network-security-config>
```

Listing B.4: Reactivating trust for user-installed CAs

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <network-security-config>
3  ...
4  <base-config>
5      <trust-anchors>
6          <certificates src="system" />
7          <certificates src="user" />
8      </trust-anchors>
9  </base-config>
10 ...
11 </network-security-config>
```

Listing B.5: Insecure NSC Snippet from the Mopub Library

```
1  <?xml version="1.0" ?>
2  <network-security-config>
3     ...
4    <base-config cleartextTrafficPermitted="true">
5      <trust-anchors>
6        <certificates src="system"/>
7      </trust-anchors>
8    </base-config>
9    <domain-config cleartextTrafficPermitted="false">
10     <domain includeSubdomains="true">example.com</domain>
11     <domain includeSubdomains="true">cdn.example2.com</domain>
12   </domain-config>
13 </network-security-config>
```