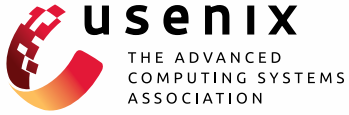

To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections

Marten Oltrogge and Yasemin Acar, ***Leibniz Universität Hannover;*** **Sergej Dechand and Matthew Smith,** ***Universität Bonn;*** **Sascha Fahl,** ***Fraunhofer FKIE***

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/oltrogge

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

To Pin or Not to Pin
Helping App Developers Bullet Proof Their TLS Connections

Marten Oltrogge, Yasemin Acar
DCSEC, Leibniz University Hannover
oltrogge,acar@dcsec.uni-hannover.de

Sergej Dechand, Matthew Smith
USECAP, University Bonn
dechand, smith@cs.uni-bonn.de

Sascha Fahl
FKIE, Fraunhofer
fahl@fkie.fraunhofer.de

Abstract

For increased security during TLS certificate validation, a common recommendation is to use a variation of pinning. Especially non-browser software developers are encouraged to limit the number of trusted certificates to a minimum, since the default CA-based approach is known to be vulnerable to serious security threats.
The decision for or against pinning is always a trade-off between increasing security and keeping maintenance efforts at an acceptable level. In this paper, we present an extensive study on the applicability of pinning for non-browser software by analyzing 639,283 Android apps. Conservatively, we propose pinning as an appropriate strategy for 11,547 (1.8%) apps or for 45,247 TLS connections (4.25%) in our sample set. With a more optimistic classification of borderline cases, we propose pinning for consideration for 58,817 (9.1%) apps or for 140,020 (3.8%[1]) TLS connections. This weakens the assumption that pinning is a widely usable strategy for TLS security in non-browser software. However, in a nominal-actual comparison, we find that only 45 apps actually implement pinning. We collected developer feedback from 45 respondents and learned that only a quarter of them grasp the concept of pinning, but still find pinning too complex to use. Based on their feedback, we built an easy-to-use web-application that supports developers in the decision process and guides them through the correct deployment of a pinning-protected TLS implementation.

1 Introduction

Android is the major platform for mobile users and mobile app developers. Many apps handle sensitive information and deploy the transport layer security protocol (TLS) to protect data in transit. Previous research uncovered security issues with TLS in mobile apps [7, 8, 9, 2, 22] 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 [24, 18]: Adopted from web-browsers, default TLS certificate validation relies on a huge number of root CAs pre-installed on all Android devices [24]. Hence, all Android apps suffer from the same issues as web-browsers: A single malicious CA is able to conduct Man-In-The-Middle attacks (MITMAs) against all apps trusting the respective certificate. To make things even worse, Fahl et al. [8] 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 [1, 14, 17, 8]. We use the term *pinning* in this paper 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

[1] This smaller percentage in the optimistic case is caused by a different prevalence of third party library use.

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 paper, we 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.

2 Background and Motivation

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). When attempting to establish a secure connection, the server provides the client with a *certificate chain*.

2.1 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. 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 [3]; 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 [20]) for which the certificate was issued.

Further checks Complete certificate validation may include further checks, e. g. certificates have to be checked for *revocation* which is done via a certificate revocation list (CRL)[3] or Online Certificate Status Protocol (OCSP)[21].

2.2 MITM Attack

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.

2.3 Alternative Validation Strategies

Over the years, alternative certificate validation strategies have come up [12, 26, 16, 15, 10, 17]. While they all provide approaches to check a certificate chain's validity, our paper focuses on *pinning* [17] as one of the most recommended alternative strategies for non-browser software.

2.3.1 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 [17]. Required parameters to be pinned need to be available in an application before the TLS handshake happens. 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.

2.3.2 Trust on First Use (TOFU)

Another notion of pinning is *trust on first use* (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 2.3.1) and the default validation strategy. TOFU is used to secure SSH connections and is an opt-in feature for HTTPS webservers for which it is called HTTP Public Key Pinning (HPKP [5]). As of today, HPKP has not found widespread adoption on the web [11].

2.4 Flaws in client implementations

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 [18]. Additionally, the Android framework enables developers to provide custom implementations for handling certificate validation. 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
- and custom implementations are required to implement leaf or CA pinning.

2.5 Related Work

Recently, TLS has been subject to urgent flaws. Studies by Fahl et al. [7] and Georgiev et al. [9] uncovered a disastrous state of TLS-Code. Georgiev et al. [9] show numerous flaws in non-browser TLS-Code. Fahl et al. [7] identify numerous applications featuring erroneous custom TLS-code that potentially renders applications vulnerable to MITMAs. This is caused by incorrect implementations of custom certificate handling. Investigations on the reasons illustrate that especially developers without a security focus are unaware of the correct use of the APIs that Android provides: They carelessly incorporate code snippets from platforms such as `stackoverflow.com` [8]. In particular, self-signed certificates used during development lead to erroneous code that makes applications vulnerable by deactivating certificate validation completely.

Both of the above investigations [7, 9] focused on the difficulties experienced by developers with the implementation of correct certificate validation. However, they discuss possible countermeasures rather sketchily and do not evaluate the applicability of pinning for non-browser software.

To improve the state of erroneous certificate handling and to mitigate the threats, Fahl et al. [8] suggest to completely disallow custom code. Instead, they propose a framework that allows to realize common use-cases (e.g. self-signed certificates for testing, pinning, default warning messages, etc.) by configuration without the developer writing any TLS-related code. Tendulkar and Enck [23] suggest a similar approach. However, although both approaches try to improve the usability of certificate handling for software developers, they do not support developers in the decision process of whether pinning is a recommendable certificate validation strategy.

3 An Exit Strategy

Previous work shows that TLS is complex and error-prone. 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 [17, 14, 1, 8]. 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.

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`[2]. 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.

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` [7] 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 [13] 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. [8] 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. [8].

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:

[2] `https://code.google.com/p/android/issues/detail?id=76501`

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 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.[3]. 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 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.

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 2.3.1).

CA Pinning CA pinning effectively limits the number of trusted CAs (cf. Section 2.3.1). We treat conventional pinning (cf. Section 2.3.1) and TOFU (cf. Section 2.3.2) equally, since both provide a similar level of security (cf. Section 2.3.1) 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.

[3] Administrators can configure HTTP Public Key Pinning (HPKP) 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).

4.2 Classification Details

Algorithm 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 use the default strategy, since ownership of the relevant API is not guaranteed.

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.

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:

4.3.1 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 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 1: Relevant API Calls.

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.

4.3.3 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

Algorithm 1: The Classification Process.

```
for r ∈ results do
    dependencies ← dependencies(r);
    set strategy as default;
    if dependencies ≠ ∅ then
        if ∃d ∈ dependencies|d ∈ {exposed intent, UI − Component} then
            continue;
        else if ∃d ∈ dependencies|d ∈ {unexposed intent, variable, configuration} then
            continue;
    if not callInLibrary(r) and not isPublicHost(r) then
        host ← host(r);
        schema ← schema(r);
        cert ← cert(r);
        dependencies ← 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;
```

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.

4.3.4 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.

5 Implementation Details

To decide whether pinning for a TLS connection is advisable and to address the above challenges (cf. Section 4.3), we implement our classification strategy in a multi-step process. We extend[4] the MalloDroid tool [7] and execute the following steps:

1. Disassemble a given Android application to gain access to the application's code and call graph (cf. Section 5.1).
2. Identify relevant API calls – i.e. API calls that implement TLS connections in apps (cf. Sections 5.2 and 4.3.1).
3. Extract information for remote origins applying program slicing (cf. Sections 5.3 and 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 4.2 and 5.4).

[4]Our MalloDroid extension is available at `https://github.com/sfahl/mallodroid`.

5.1 Disassembly

In Step 1, we use androguard [4] to disassemble apps and construct call graphs for further processing.

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 4.3).

We consider API calls as *relevant* if they are used during the process of TLS connection establishment (cf. Table 1).

5.3 Program Slicing

In Step 3, we apply backwards program slicing [25] 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. [19], 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.

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 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 2: StringBuilder Analysis

```
Data: stringBuilderObjects sbos
Result: new originSubstring O
begin
    for sb ∈ sbos do
        methodInvocations ←
        methodsOnInstance(sb);
        originSubstrings ← ∅;
        origin ← null;
        for mi in 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`.

5.4 Decide on Validation Strategy

In the final Step 4, we assess whether API and/or origin ownership is given (cf. Section 4). For pinning candidates, X.509 certificate information is collected and a decision for or against pinning is made (cf. Section 4.2).

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.

6 Evaluation

We applied the classification algorithm (cf. Section 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 2) as well as on a per-app base (see Figure 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). Altogether we found 20,020,535 calls to network related API calls (cf. Table 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 4.2) gives detailed insights into the applicability of pinning. Table 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 2: Distribution of Network API Calls.

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 4.2, 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 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

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 3: Top 10 Third Party Libraries.

connections without any interaction with an app's custom code.

We found the `AndroidPinning` library[5] 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 3.2).

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.

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.

[5] `https://github.com/moxie0/AndroidPinning`

	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.

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: Origins in Custom Code – Connections marked with 📌 can be pinned.

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 used by apps of a single developer only (cf. Section 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). This affects 99,996 of the TLS connections we identified in our data-set (cf. Table 4a).

Table 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 con-

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 5: Top 10 Remote Origins.

trol 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). This affects 45,247 TLS connections in our data-set (cf. Table 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 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 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 6: X.509 Certificates Statistics.

We recommend pinning for all of these connections (cf. Section 4). We recommend leaf pinning for the connections that use a self-signed certificate and CA pinning for all other connections (cf. Table 7).

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

Table 7: Pinning Statistics (Conservative).

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). 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), 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), 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 con-

nections 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.

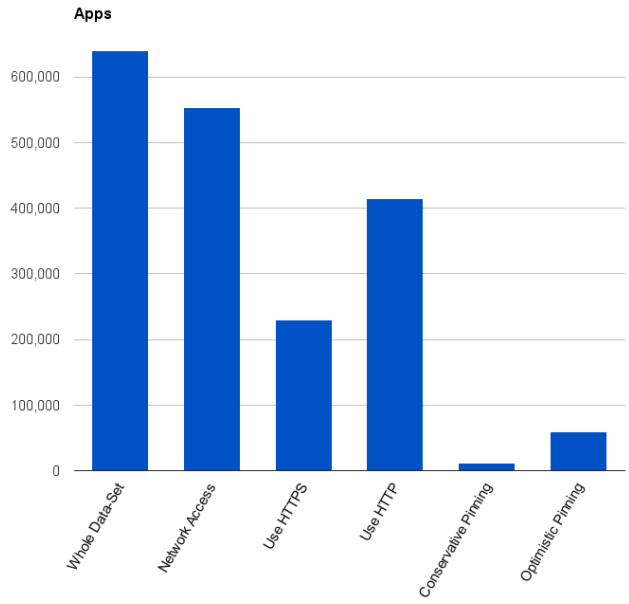

Figure 1: Statistics and Classification Results for Apps; *Network Access* includes apps with custom-coded, library- and dynamic HTTPS and HTTP connections.

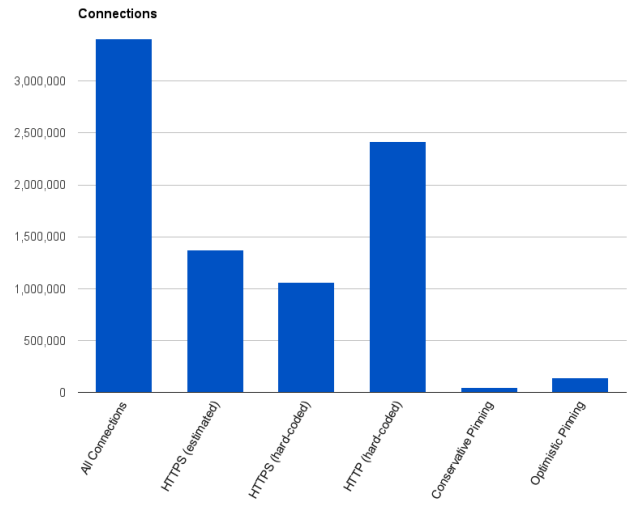

Figure 2: Statistics and Classification Results for Connections.

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.

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 [6].

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 3 illustrates the average apps update periods.

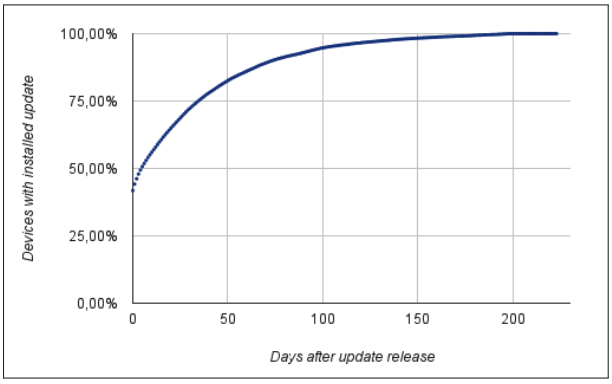

Figure 3: Average days between the release of a new app version and its installation.

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 [6].

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.

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

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.

7.1 Feedback

We analyzed all apps in our sample set and extracted possible candidates for pinning [7]. 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[8] 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.

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

[8] 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.

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.

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 [9] 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.

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

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.

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/`, 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) 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).

Finally, the strategy classification is conducted and we present the developer with our recommendation for making their connections as secure as pos-

sible. 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 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.

8 Limitations

In addition to the limitations described in Section 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.

9 Conclusions

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 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.

References

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

[2] Bates, A., Pletcher, J., Nichols, T., Hollembaek, B., Tian, D., Butler, K. R., and Alkhelaifi, A. Securing ssl certificate verification through dynamic linking. CCS '14, ACM, pp. 394–405.

[3] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `https://datatracker.ietf.org/doc/rfc5280/?include_text=1`, May 2008.

[4] Desnos, A. Androguard. `http://code.google.com/p/androguard/`.

[5] Evans, C., Palmer, C., and Sleevi, R. Public Key Pinning Extension for HTTP. `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-21`, Oct 2014. Internet-Draft.

[6] 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. CCS '14, ACM, pp. 1143–1155.

[7] 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. CCS '12, ACM, pp. 50–61.

[8] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. CCS '13, ACM, pp. 49–60.

[9] 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. CCS '12, ACM, pp. 38–49.

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

[11] Kranch, M., and Bonneau, J. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning.

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

[13] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. Chex: Statically vetting android apps for component hijacking vulnerabilities. CCS '12, ACM, pp. 229–240.

[14] Marlinspike, M. Android Pinning. `https://github.com/moxie0/AndroidPinning`.

[15] Marlinspike, M. TACK: Trust Assertions for Certificate Keys. `http://tack.io/draft.html`.

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

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

[18] Perl, H., Fahl, S., and Smith, M. You Won't Be Needing These Any More: On Removing Unused Certificates From Trust Stores. In *Financial Cryptography and Data Security 2014* (2014).

[19] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code.

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

[21] 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.

[22] 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.

[23] Tendulkar, V., and Enck, W. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities.

[24] Vallina-Rodriguez, N., Amann, J., Kreibich, C., Weaver, N., and Paxson, V. A tangled mass: The android root certificate stores. CoNEXT '14, ACM, pp. 141–148.

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

[26] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving ssh-style host authentication with multi-path probing. ATC'08, USENIX Association, pp. 321–334.