TagDroid: Hybrid SSL Certificate Verification in Android ⋆

Hui Liu, Yuanyuan Zhang, Hui Wang, Wenbo Yang, Juanru Li, and Dawu Gu

Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
yyjess@sjtu.edu.cn

Abstract. SSL/TLS protocol is designed to protect the end-to-end communication by cryptographic means. However, the widely applied SSL/TLS protocol is facing many inadequacies on current mobile platform. Applications may suffer from MITM (Man-In-The-Middle) attacks when the certificate is not appropriately validated or local truststore is contaminated. In this paper, we present a hybrid certificate validation approach combining basic certificate validation against a predefined norm truststore with ways by virtue of aid from online social network friends. We conduct an analysis of official and third-party ROMs. The results show that some third-party ROMs add their own certificates in the truststore, while some do not remove compromised CA certificates from the truststore, which makes defining a norm truststore necessary. And the intuition to leverage social network friends to validate certificate is out of the distributed and "always online" feature of mobile social network. We implemented a prototype on Android, named TAGDROID. A thorough set of experiments assesses the validity of our approach in protecting SSL communication of mobile devices without introducing significant overhead.

Keywords: MITM attacks, SSL certificate verification, Mobile SNS

1 Introduction

SSL is widely used to secure communications over the Internet. To authenticate the identity of communication entity, X.509 certificates and hence Public Key Infrastructure (PKI) are used. In PKI system, an X.509 certificate binds a public key with a particular distinguished name, and it certifies this relationship by means of signatures signed by the Certificate Authority (CA). Therefore, to establish a secure connection between client and server, i.e. to prevent MITM attacks, a client must verify the certificate received from the server to guarantee

⋆ Supported by the National Science and Technology Major Projects (2012ZX03002011-002), National Key Technology Research and Development Program of theMinistry of Science and Technology of China (2012BAHXXXXXX) and the National Natural Science Foundation of China (No.61103040).

the following two critical points: 1) the received certificate is issued for the server (distinguished name matches the hostname); 2) the CA who signed the certificate is in the predefined set of trusted root CAs, which is known as *truststore*.

While in traditional desktop environment, SSL is mostly used in Web browsers and its implementation is taken good care of by several large corporations, it is noticeable that, with the proliferation of Android devices and thereby prominent emergence of Android apps in recent years, SSL is used more and more to secure communications in varieties of apps developed by millions of developers. However, out of different reasons ranging from wanting to use self-signed certificates during development to being unaware of security implication of accepting any certificate [9], the developers fail to correctly implement the certificate verification when using SSL, which gives rise to the insecure communication.

Besides, there still exists issues even if the certificate verification is correctly carried out. On one hand, the truststore is a critical part. For Android, the truststore that comes along with the system can be manipulated, which is a potential threat. If a malicious root CA is trusted, all the certificates it issued will be accepted. On the other hand, there might be some severs that provide certificates signed by untrusted root CAs. It's been found that SSL/TLS communications without a valid certificate (e.g., self-signed certificate provided by the server) are quite common. A study in [5] found that the false warning ratio is 1.54% when examining 3.9 billion TLS connections, which is an unacceptably high rate since benign scenarios are orders of magnitude more common than attacks. And for users, when faced with a self-signed certificate, they hardly have a clue to tell whether it is from the legitimate server or it is replaced by a MITM attacker.

Existing approaches to dealing with these problems are introduced in Section 6 and their limitations are depicted. While at the same time, we notice that the "always online" feature of smart mobile device is becoming the most popular vector of social network service (SNS). Emerging works leverage the distributed nature of SNS in file sharing [6] [10], poll/e-voting and other scenarios.

In this paper, we attempt to deal with the server authentication issue in Android through a way that combines the basic certificate validation with ways by virtue of online social network friends helping verify the certificate. It is carried out in three steps. Firstly, we aim to secure communications even if the apps are not correctly developed, since a large number of apps that fail to validate certificate correctly have already been installed on massive number of devices. Thus, we perform as a friendly proxy that takes care of all the SSL connections, and for every connection, we perform a strict certificate validation. Secondly, to prevent truststore from being manipulated, we define a *norm truststore*. The strict certificate validation is carried out against the norm truststore such that only the certificates issued by CAs in the norm truststore are accepted. Lastly, for the certificates that fail to pass the strict check or are not issued by CAs in the norm truststore, we launch a collaborative certificate verification to seek for extra information to decide whether it is really presented by server or it is a malicious certificate. We consult to social network friends for help, asking

them to fetch certificates from the same server and provide a credibility indicator about the certificate.

We implement a prototype named TagDroid and conduct experiments to evaluate the effectiveness and overhead. The results indicate that this scheme is effective in defeating MITM attack and dealing with self-signed certificates without significant overhead.

2 Background

2.1 Android & SSL Certificate Verification

When authenticating a server, two key parts must be guaranteed: the certificate is signed by a trusted source, and the server talking to presents the right certificate. The Android framework takes care of verifying certificates and hostnames and checking the trust chain against the system truststore. By simply including the two lines below, a client can launch a secure connection.

```
URL url = new URL("https://example.org");
URLConnection Conn= url.openConnection()
```

If a certificate is not issued by CA in truststore, e.g. a self-signed certificate, or the DN does not match hostname, an exception will be thrown out.

However, prior work [8] indicates that 8.0% apps are not correctly implemented and thus vulnerable to MITM attacks, and a deep analysis shows that these apps use "customized" SSL implementations that either accept all certificates or accepts all hostnames.

2.2 Android & TrustStore

The Android framework validates certificates against the system truststore by default. In pre-ICS (Ice Cream Sandwich, Android 4.0), the truststore is a single file stored in /system/etc/security/cacerts.bks, and is hard-wired into the firmware. User has no control over it, but if the device is rooted, a keytool can be used to repackage the file and replace the original one. This kind of operation may not be possible for attackers or malicious apps to leverage. However, the truststore comes along with the firmware, and now many third-party ROMs are available and widely installed. We conduct a detail analysis on truststores from both third-party and official ROMs, which is described in 4.2. The results indicate that a third-party ROM that has been downloaded more than 2,000,000 times is found to have added 3 extra certificates compared with the official one of same version. According to the result, we can reasonably assume a malicious third-party ROM maker could deliberately add a root CA into the truststore, via which he could launch MITM attacks successfully. And devices having installed this malicious ROM would have no chance to find out something is going wrong.

Since Android 4.0, user can add and remove trusted certificates, which gives user more power in controlling the truststore. Users are provided with an option

"Settings > Security > Install From Storage" to add root certificates, with no root permission required but needing pin lockscreen on. The other way of changing truststore is to directly copy the certificate to the directory /etc/security/-cacerts with root permission. The truststore is under threat as well. Apart from the factor of third-party ROMs, an application with root permission granted can successfully add a certificate to the cacerts directory without any prompt coming out.

Furthermore, as Android version evolves, some root certificates are removed for some reason (for example, a compromised CA). But devices with old version installed still trust those certificates and therefore ones signed by them. Attackers can make use of this weakness with little effort.

An effective approach for apps to defeat attacks mentioned above is to initialize a TrustManagerFactory with their own keystore file that contains the server's certificate or the issuing certificate, which is called pinning [11]. However, this is only suitable for those who do not need to connect to practically every possible host in the Internet. By using pinning, no matter what happens to system truststore, there will be no effect on these apps.

Another issue concerning validating certificate against truststore is that when a user connects to a server whose certificate is issued by private CAs, a warning is prompted asking whether to proceed. But the user will see the same warning when the received certificate is signed by the MITM attacker who fails to manipulate user's truststore. Therefore, under this condition, there is no extra information to help the user make further decision.

3 Attack Scenario

In this section, we present the type of attacks we aim to tackle with. We only consider the local MITM attacks. As shown in Fig. 1. The attacker (Mallory) is located near the client (Alice) and replaces the certificate (CertB) sent by server (Bob) with its own certificate (CertM). If Alice accepts CertM, all messages exchanged between Alice and Bob will be plaintext to Mallory. According to the certificate Mallory presents to Alice, we define the following two attack scenarios.

3.1 Self-signed Certificate

Mallory simply uses a self-signed certificate with the property that either DN does not match hostname or it is not issued by CAs trusted by Alice. This is the easiest kind of attack to carry out, and apps having correctly implemented the verification using standard APIs will not be vulnerable to this attack.

However, as mentioned in 2.1, lots of apps using customized implementation are not able to take care of this verification. We tested some of the most popular apps in China, using BurpSuite to replace the certificate sent from the server with a self-signed certificate, and find out there indeed are apps that don't validate the certificate and all the traffic can be decrypted.

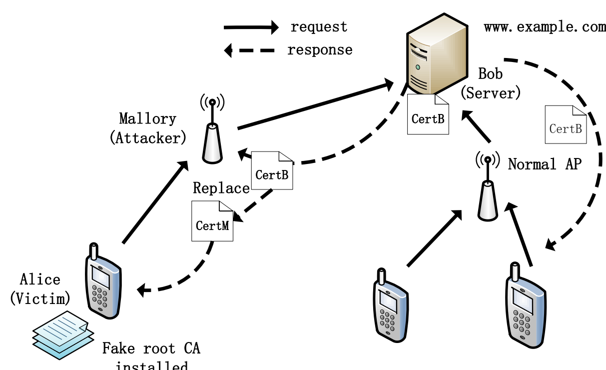

Fig. 1. Attack Scenario. Local attacker Mallory between Alice and Bob replaces certB sent from Bob with certM. Alice may have installed a fake root CA such that Mallory can decrypt most of Alice's secure communication even if correct certificate verification is carried out.

3.2 Installed Malicious Root CA-signed Certificate

In this scenario, Mallory manages to have root CA installed in client's truststore, so certificates signed by this pre-installed root CA are trusted by apps validating certificates in the default way. This attack is much more difficult since Mallory needs a certificate installed in user's truststore in advance. However, third-party ROMs pave the way for this kind of attacks (see 4.2).

As mentioned in 2.2, this kind of attack can be avoided by using certificate pinning if apps only need to connect to a limited number of servers. Pinning performs good but seems to be not widely adopted yet since lots of apps are found vulnerable.

We conduct the experiment by installing the root CA of BurpSuite in the truststore that signs all the certificates. Some apps related to banking and payments are found to be vulnerable to this kind of MITM attacks.

4 TagDroid

To correctly authenticate a server, there are three aspects to meet:

- guarantee all the certificates to be accepted are for the right server from a trusted source. This security can be provided by the framework, either the Android framework or the web/application framework when implementing correctly.
- constrain the trusted source to a definite set, which will assure the truststore is not contaminated.

– able to tell whether it is from the server or a malicious MITM attacker when a self-signed certificate or a private CA issued certificate is received.

Out of these three reasons, we propose our TagDroid, which aims to protect all apps involving secure communication from MITM attacks. In this section, we will take an overview of TagDroid, and elaborate on the two modules which respectively take care of validating certificates against the norm truststore correctly and keeping everything going when exceptions happen.

4.1 Overview

TagDroid takes care of the secure communication traffics in and out of the system, and its first module, named ValMod, validates the received certificate against local truststore by checking 1)whether it is issued for the right server the app want to talk to and 2)it is issued by a trusted source.

The first check is completed by using the standard API that Android framework supplies, while the norm truststore is what we constructed by comparing all the certificates included in different versions of Android and different ROMs widely used. The second module comes into effect when the received certificate fails to pass ValMod. This can happen considering that we have constrained the truststore and there are servers using self-signed certificate. We consult to social network friends for help at this point by asking what the certificate is from their perspectives, and this procedure is called PeerVerify. The high level overview of TagDroid is shown in Fig. 2.

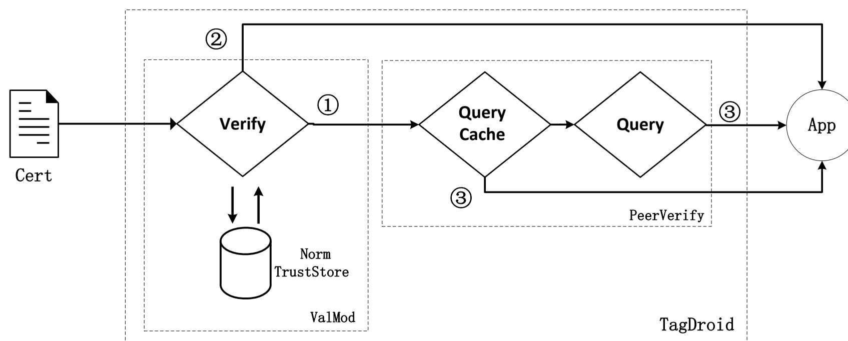

Fig. 2. Architecture of TagDroid. All certificates received are passed through TagDroid first. (1) Cert fails to pass the ValMod validation against norm truststore, so it is passed to PeerVerify for further verification. (2) Cert succeeds to pass ValMod so that it is directly send to apps. (3) If Cert fails to pass PeerVerify, it is discarded and the corresponding app is alerted; Otherwise, it is directly send to apps.

4.2 ValMod

The main task for ValMod is to perform a strict certificate validation against the norm truststore for each received certificate.

To decide which root CA certificate to be included in the norm truststore, we conduct an analysis of truststores from different versions of official [2] and third-party ROMs. The result is shown in Table 1.

Table 1. The number of certificates in official and third-party ROMs.

Version	2.3.5	2.3.7	3.2.4	4.0.3	4.0.4	4.1.2	4.2.2	4.3.1	4.4.2
Official	128	127	132	134	134	139	140	146	150
HTC	128	128	-	164	170	139	-	-	150
SONY	127	105	-	137	134	142	-	-	150

For official ROMs, we can see that the number increases as version evolves. Each change has some certificates removed and others added [3]. It is worth to mention that from 2.3.5 to 2.3.7, the certificate removed is DigiNotar, who confirmed to have a security breach in March, 2011, which resulted in the fraudulent issuing of certificates. For third-party ROMs, HTC 2.3.7 did not remove DigiNotar. This ROM has been downloaded 270,000 times by far and this data is only from one website. While in version 4.0.3 and 4.0.4, HTC added lots of certificates. On the other hand, SONY added its own certificates in 4.0.3 and 4.1.2, and removed 23 certificate in 2.3.7.

We have no idea why these certificates are added or removed. But this is persuasive to make clear that third-party ROMs can add or remove certificates as they want. This condition makes it necessary for us to define a norm truststore when we enforce our own validation. What's more, the update of ROM poses a problem on lots of devices. With old version ROM installed, passing the certificate verification against system truststore is not enough to guarantee secure connection.

Therefore, we define the norm truststore as all the certificates included in the official ROM of newest version (for now, it is version 4.4.2), and ValMod takes care of correctly validating certificate against this norm truststore. Certificates that succeed to pass the validation is guaranteed to be genuine, while those fails to pass are not definitely forged since there are servers using certificates that are self-signed or issued by untrusted CA.

4.3 PeerVerify

When certificates are not trusted by ValMod, there still are chances that the client receives the genuine certificates. So we need extra information to help confirm these certificates and refuse malicious certificates at the same time. We consult to social network friends by sending them the website address and asking which certificates they receive. If most of the responses contain the very

certificate that we receive, we can make the positive decision. Otherwise, we believe the certifiate we receive is not trustworthy. We assume friends will report honestly and communications among friends are secured by the social network application, for example Gtalk. We turn to social network friends for help out of the following reasons: 1) on mobile platform, friends are much more likely to be online and the online status can be stable for a long time. 2) instead of consulting a dedicated server, we believe seeking help from friends can defeat denial of service attacks. Besides, in terms of privacy, we observe that a dedicated server could raise a large number of requests("bigdata") which may be used to dig deeper information, while the number of requests sent to each friend are quite small since the scheme described below can make the requests evenly distributed on average .

PeerVerify can flexibly handle how many friends to ask and whom to ask. The workflow consists of three phases: *initialization*, *chained querying*, and *feedback analysis*. To make things clear, we define some terms here. A *session* indicates the whole process which starts from the time when a user makes a query and ends at the time when the user gets corresponding responses. A user who starts a session is called an *initiator*, and we call each involved friend as a *peer*. During initialization phase, the initiator picks a number called *expectation* to indicate how many friends it wants to consult. And the message that peer sends back contains a value called *reputation* indicating the confidence of the result.

An example shown in Fig. 3 shows how PeerVerify works. In the initialization phase, initiator picks expectation 10, chooses 3 peers with the assigned expectation 4, 4 and 2 respectively and sends the request message. In the chained querying phase, those chosen peers who receive the message do the same thing as if they are an initiator who picks 3, 3 and 1. A peer who receives request with expectation 1 does not query his friends and just sends the result back. When peers receive response message, they make some analysis and send upward until the real initiator gets the response, and this is the feedback phase.

We give further details in the rest of this part.

Initialization The main task of initialization is to decide how many peers to ask and whom to ask. Then sends the request message containing a url and *expectation* n to chosen peers. Geographically distant peers are chosen with high priority and are assigned with larger expectation, which could largely avoid peers being in the same attack area and thus providing useless information. The choice of n relies on the number of online friends, the network distance, the quality of the network connection etc. And users can impact on the choice of n by setting security option to different levels, with each level standing for a value range within which n is picked.

Chained Querying The basic chained query idea is to recursively pass down the query from one to another. For a peer who are assigned the expectation of n_b, he expects aids from $n_b - 1$ peers in total, so he split $n_b - 1$ into $n_{b1} + n_{b2} +$

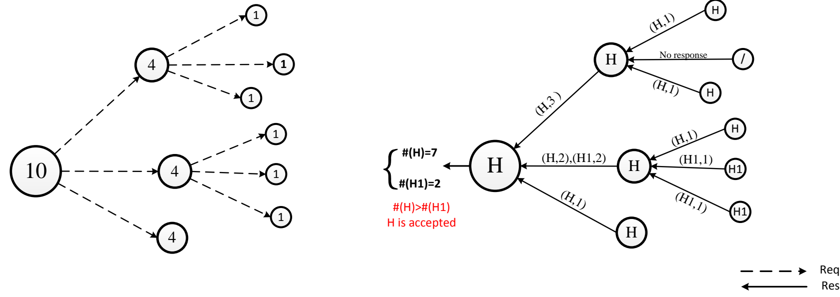

Fig. 3. An example that illustrates how PeerVerify works. (Left) Chained querying phase that recursively pass down the expectation. (Right) Feedback analysis phase that peers collect and process responses, and initiator correspondingly makes decision.

$\ldots + n_{bi} + \ldots + n_{bx}$, in which x indicates the number of peers he chooses and n_{bi} indicates the expectation assigned to each peer.

Feedback Analysis After the recursive query procedures, peers send back the feedbacks according to the query message they've received. By collecting and comparing the received feedbacks, peer starts a **reputation calculation** process. For example, initiator Alice receives the responses from peer Bob ($set\{(H_1, r_{b1})$ $, (H_2, r_{b2})\}$) and Charlie ($set\{(H_1, r_{c1}\}$), and Alice herself gets the hash H_1. H stands for the certificate hash and r stands for the reputation. Alice calculates the set union $set\{(H_1, r_{b1}),(H_2, r_{b2})\} \cup set\{(H_1, r_{c1})\} \cup set\{(H_1, 1)\} = set\{(H_1, r_{b1}{+}r_{c1} + 1), (H_2, r_{b2})\}$. If she is initiator, she compares the value of $r_{b1} + r_{c1} + 1$ with r_{b2} to decide accept H_1 or H_2. If not, she just sends the calculated set upwards.

In all, after PeerVerify, initiator can get a certificate hash H that trusted by most peers. If the hash of the certificate that initiator receives equals H, TagDroid will accept the certificate. Otherwise, it considers the certificate as malicious and terminates the connection.

5 Evaluation

We implemented a TagDroid prototype on Android platform, and its PeerVerify module piggybacks on the popular IM application Google Talk. Smack, an Open Source XMPP (Jabber) client library [4], is used to build the customized Google Talk client. The customized client also records how many query it has launched, to evaluate the performance of ValMod's norm truststore.

We built a *virtual social network* environment which simulates the chained querying and feedback, to evaluate the time latency and communication overhead

brought by PeerVerify. This environment is set up by creating a $Watts-Strogatz$ [14] small world graph in Python, which is generated with parameter($N,\ k,\ p$), meaning N nodes forming a ring and each node connects with its k nearest neighbors. For each edge u-v, it has a probability of p to be replaced with a new edge u-w, and w is randomly chosen from the existing nodes. Each node stands for a single user, and the edge connecting node ($u,\ v$) stands for the friend relationship. Each node is assigned the ability to divide expectation in a way that, for a initiator who has f *long-distance peers* and chooses expectation n, it randomly divides n into x parts with x randomly chosen from $[1,\ f]$. Then it distributes these x parts to x long-distance peers. For a node with ordinal i, long-distance peers are defined as nodes with ordinal j that $|i-j| > k/2$.

5.1 Effectiveness

Since untrusted CA issued certificates will definitely fail to pass ValMod, these questionable certificates are all sent to PeerVerify. We conduct our experiment with TagDroid installed on SamSung Note 3 under an attack environment that replaces all the certificates with ones that issued by an untrusted CA. When TagDroid does not take effect, apps that are vulnerable to self-signed attack accept the forged certificate and others will reject since they validate certificates against system truststore. When having the untrusted CA installed in system, only a few reject the forged certificates because of pinning. While with TagDroid taking effect, these forged certificates are all rejected.

5.2 Performance

Communication overhead All the extra communications are brought by PeerVerify. And the total communication overhead increases proportionally with the expectation n chosen by the initiator. The payload $S_{payload}$ generated by a complete query session can be approximately calculated as $S_{payload} = n * S^{ij}_{payload}$, $S^{ij}_{payload}$ stands for the payload generated by a *smallest session* that peer j helps peer i validate the certificate with peer j asking no more peer for help. $S^{ij}_{payload}$ contains two part: the Google Talk traffic that generated by communication of peer i and j, and the SSL traffic that j generated when communicating with the server to get the certificate. By running the TagDroid client installed on two devices that connect to the same monitored hotspot, we can capture the whole traffic. By adding these traffic up, $S^{ij}_{payload}$ amounts to about 8470 bytes. This data may vary since the length of content field in query and response message is different for different URL and expectation n.

Latency The time period of a complete and successful PeerVerify session is regarded as the latency in our evaluation. It begins from the initiator sends out the query, and ends in that the trust decision has been made. The *latency* of a complete session can be calculated by the longest hops h and latency $latency_{ij}$ generated by a smallest session with $latency = h * latency_{ij}$. By conducting

a smallest session on TagDroid client, we can see the $latency_{ij}$ is about 1s. Therefore, *latency* varies with h. h is related with expectation n and the decision of each peer about how to divide and distribute n. To get a general idea of h, we conduct an experiment in simulation environment. The results show that, given network size, h is related with *expectation* n and the number of friends k in the list. With the increasing of expectation, the latency (max hops h) increases as well. However, the larger k is, the slower latency increases, and when k is large enough($k > 20$) and expectation is less than 30, latency will not decrease with growing k. Besides, peer only waits for response for a limited time t, so the total time will not be larger than $h * t$.

6 Related Work

The existing approaches to enhancing CA-based certificate authentication can be classified into two categories, either by relying on existing architectures such as DNS or PGP, or by introducing notary [12] [15] [1] to provide reference information. But they all have their limitations. For DNS based approach, widely deployment is challenging. Others have issues such as limited number of notaries, requiring server cooperation, etc. For proposals that aim to fix implementation issues in Android applications, they either provide developers with easy-to-use APIs [9], or reference extra information provided by a specific server [7]. The former scheme requires system modification, which may have deployment issue. The latter consults to a fixed server, making the security of the whole scheme rely on the security of a specific server, and having only one entity to reference may fail to provide enough information to help user make the right decision. For approaches to dealing with certificate verification problems in general, [13] proposes a social P2P notary network, which uses advanced techniques (such as secret sharing, ring signatures and distributed hash table) to tackle privacy, availability and scalability issues. But this scheme is too much to apply on Android platform, since most Android applications only talk to a few specialized server.

7 Conclusion

Due to the lack of proper certificate verification and untrusted CAs installed in system truststore, some Android applications that use SSL protocol to secure the communication are vulnerable to the MITM attacks. TagDroid is proposed to tackle this problem. It is a hybrid verification system combining correct certificate validation against a norm truststore with a collaborative way that relies on the social friends to help notarize questioned certificates. TagDroid is capable of detecting illegitimate certificates which defeats MITM attack to a large extent. We implement a prototype called TagDroid and easily install it on the genuine Android system. It does not require any modification on the system, and just has little effect on system performance such that users can barely notice its existence. To define the norm truststore, we carry out a thorough analysis of

ROMs both from official and third party. To evaluate the performance impact of TagDroid as a network system, TagDroid is deployed on a Samsung Note 3 and a simulation is carried out in a large scaled social network with 10,000 peers. The performance analysis results show that TagDroid brings low latency and communication overhead.

References

1. Perspectives project. `http://perspectives-project.org/`.
2. Android source. `https://android.googlesource.com/platform/libcore/`.
3. Android Source Diff. `https://android.googlesource.com/platform/libcore/log/android-2.3.4_r0.9/luni/src/main/files/cacerts`.
4. Smack api. `http://www.igniterealtime.org/projects/smack/`.
5. Devdatta Akhawe, Bernhard Amann, Matthias Vallentin, and Robin Sommer. Here's my cert, so trust me, maybe?: understanding tls errors on the web. In *Proceedings of the 22nd international conference on World Wide Web*, pages 59–70. International World Wide Web Conferences Steering Committee, 2013.
6. Kang Chen, Haiying Shen, Karan Sapra, and Guoxin Liu. A social network integrated reputation system for cooperative p2p file sharing. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–7. IEEE, 2013.
7. Mauro Conti, Nicola Dragoni, and Sebastiano Gottardo. Mithys: Mind the hand you shake-protecting mobile devices from ssl usage vulnerabilities. In *Security and Trust Management*, pages 65–81. Springer, 2013.
8. Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.
9. Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.
10. Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651. ACM, 2003.
11. Langley. Public key pinning. `https://www.imperialviolet.org/2011/05/04/pinning.html`.
12. Moxie Marlinspike. Convergence. `http://convergence.io/index.html`.
13. Andrea Micheloni, Karl-Peter Fuchs, Dominik Herrmann, and Hannes Federrath. Laribus: Privacy-preserving detection of fake ssl certificates with a social p2p notary network. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 1–10. IEEE, 2013.
14. Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.
15. Dan Wendlandt, David G Andersen, and Adrian Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.