An evaluation of smartphone communication (in)security

Christoffer Brodd-Reijer

Institutionen för informationsteknologi
Department of Information Technology
Abstract

**An evaluation of smartphone communication (in)security**

Christoffer Brodd-Reijer

The purpose of this study is to examine and evaluate the security of the data traffic sent to and from smartphone devices. Since smartphones are becoming more common, are highly connected, often use cloud based computation, and contain highly personal data, it is important that the communication is secure and safe. This paper examines the Android and iOS platforms and focuses on three key parts: platform, application, and user. The platforms are evaluated on the basis of their libraries, APIs, and documentation; applications are evaluated using static code analysis and manual traffic analysis; users are examined using a social experiment. Results show that about one in twenty applications leaks sensitive data, without any difference between platforms. While the platforms do a good job educating developers about security there are room for improvements. The paper also concludes that a non-insignificant share of users are inclined to bypass important security warnings which may expose their passwords to an attacker.
# Contents

1 Introduction 3

2 Background 4
   2.1 TLS handshake ........................................ 4
   2.2 TLS sessions .......................................... 4
   2.3 Certificates ......................................... 5

3 Related work 5
   3.1 Android security ........................................ 5
   3.2 iOS security ........................................... 6
   3.3 SSL/TLS .................................................. 6
      3.3.1 Attacks against TLS ................................. 7
   3.4 Cloud security for mobile ............................... 9

4 Platforms 9
   4.1 Documentation .......................................... 9
      4.1.1 Android .............................................. 10
      4.1.2 iOS .................................................. 11
   4.2 Libraries ................................................ 13
      4.2.1 Android .............................................. 13
      4.2.2 iOS .................................................. 13
   4.3 Sample code .............................................. 14
      4.3.1 Android .............................................. 14
      4.3.2 iOS .................................................. 14
   4.4 TLS support .............................................. 15
   4.5 Summary .................................................. 15

5 Applications 15
   5.1 Selection ................................................ 16
   5.2 Test setup .............................................. 16
      5.2.1 Code analysis ....................................... 17
      5.2.2 Traffic analysis ..................................... 18
   5.3 Results .................................................. 19
      5.3.1 Traffic protection ................................... 19
      5.3.2 Certificate verification ............................. 20
      5.3.3 Vulnerable TLS traffic .............................. 21
      5.3.4 Indication of security ............................... 21
   5.4 Summary .................................................. 22

6 User behavior 23
   6.1 Connecting to open network ............................. 23
   6.2 Confrontation with invalid certificate ................. 24

7 Conclusions 25
   7.1 Platforms .............................................. 25
   7.2 Applications ............................................ 26
   7.3 Users .................................................... 26
   7.4 Future work ............................................ 27

8 References 27

Appendix A Graphs 31

Appendix B Tables 33
1 Introduction

Smartphones today contain highly personal data, are location aware, can run third party software from thousands of vendors, and are carried with us almost every hour of the day. They are often used in combination with cloud services where data resides, and may even be processed, on foreign servers. This makes it important for the devices to provide good security for the user. Miller predicts that smartphones will become a more popular target by attackers than Personal Computers in a few years [21].

There has been several disclosed vulnerabilities which affect smartphones. A security advisory from Microsoft [35] describes a vulnerability in Windows Phone 7.8 and 8 which allows an attacker to steal domain credentials by exploiting PEAP-MSCAPv2 after tricking a user to connect to a rogue Access Point. By accessing the credentials of a user, an attacker can impersonate the user and access other protected systems. Attackers can thus use smartphones as a step in mounting a larger attack on more sensitive systems.

Alex Klyubin, Android Security Engineer, recently disclosed [20] a vulnerability in the Java Cryptography Architecture used by Android. The generator of random numbers is used, among other things, to create signatures for certificates. The vulnerability makes these random number predictable by an attacker. Klyubin noted that the flaw has been used by attackers to compromise users’ Bitcoin wallets and steal money. This highlights that the tools available to developers are important in ensuring the security of applications which users trust with their sensitive data.

While it is hard for individual developers to protect against vulnerabilities in the platform, they can still take precautions to secure the data that the app sends over the network. Unfortunately one of the most common security mistakes made by developers of smartphone application is either improper use of traffic encryption or not using any protections at all [8].

This paper looks at the communication to and from a smartphone device and evaluates it from a security standpoint. Data being communicated can be of various sensitivity and can be protected by various mechanisms. Confidentiality prevents an adversary from reading data, authenticity ensures the communicating parties can verify each other’s identities, and integrity prevents tempering and manipulation of data by a third party.

Most web based cloud systems use the HyperText Transfer Protocol (HTTP) to communicate with devices. This protocol is insecure as it transmits data in plain text. HTTP can be made more secure by combining it with Transport Layer Security (TLS) protocol, or its predecessor Secure Socket Layer (SSL). This is known as the HTTPS Secure (HTTPS) protocol which provides both confidentiality and authentication for communication over HTTP.

At the heart of TLS is a certificate scheme where the entity being authenticated provides a certificate to the authenticator. The certificate is signed by an authority either directly trusted by the authenticator or in turn signed by a trusted authority. If a certificate cannot be properly validated, many implementations allow the user to make a decision to ultimately accept or refuse the connection. The weak link in secure communication over HTTPS is thus the validity of the certificates used. Developers who choose to use HTTPS to transmit data securely must therefore ensure that the certificates used in the communications are valid and correct before sending sensitive data over the channel.

This paper is divided into three parts: platform, applications, and user behavior.

First the platforms and libraries provided by the operating system are evaluated in section 4. Since Android and iOS are the two most popular smartphone operating systems today [34], focus is on those two platforms. These systems often has in place various mechanisms to protect applications from each other, and protect the operating system itself from malicious applications. In this paper only the tools which provide developers with means of creating secure communications for their applications are evaluated.

Further, a selection of applications are tested in section 5 using static code analysis and manual traffic analysis. Applications carrying both sensitive and non-sensitive data are selected and their communication is both monitored and manipulated in order to detect vulnerabilities such as disclosure of sensitive information. The applications using encryption to protect the communication are attacked using a man in the middle (MITM) attack, where an attacker intercepts the communication between the two parties in order to break the encryption and retrieve the plaintext either partly or in full.

Lastly, section 6 evaluates the behavior of users. In some cases, an app presented with an invalid certificate may not outright refuse to establish a connection but instead allow the user to make the ultimate decision whether to proceed with the connection. A possible security risk with this approach is that users may accept a false certificate without
being fully aware of the possible consequences. The last part of this thesis tests this security risk by using a rogue hotspot where users who choose to connect are presented with false certificates.

By looking at these three parts it is possible to determine how developers are aided in creating secure communication, how often applications do use secure communication, and how users react when a communication is insecure or even hijacked.

2 Background

This section will provide a quick summary of the TLS protocol and how it is used in HTTPS. Readers already familiar with this may skip this section and go directly to section 3.

In standard HTTP\(^1\) traffic both requests and responses are sent in clear text, which allows an attacker to both eavesdrop and, if positioned between the two parties, to intercept and modify the traffic, without any of the participants noticing.

To prevent this the protocol HTTPS was formed in 1994 by Netscape Communications and become a standard in May 2000\(^2\).

The HTTPS protocol is standard HTTP data over a secure TLS tunnel. The TLS protocol is the predecessor to SSL. When HTTPS was formalized, the SSL protocol became TLS\(^3\).

2.1 TLS handshake

When a client requests a resource over HTTPS it sends out a ClientHello where it specifies the highest supported version of TLS, all supported cipher suites and compression algorithms, and a random number. The server will then answer with a ServerHello which contains the chosen cipher suite, protocol version, compression method, and a random number.

The server may also send a Certificate message which contains the chosen cipher suite, protocol version, compression method, and a random number.

The server also sends out a ServerHelloDone message to indicate that the handshake is done. The client may also send out a ClientKeyExchange message depending on the cipher suite. This message contains either a PreMasterSecret, encrypted with the server’s public key, a public key, or nothing.

After all negotiations are done, both the server and the client compute a common secret using the random number and the PreMasterSecret. This common secret is called the MasterSecret.

During the second step of the handshake, the client sends a ChangeCipherSpec message which indicates that all subsequent communication will be encrypted and authenticated (depending on the chosen cipher suite). Lastly the client sends the server a Finished message. The server then proceeds to do the same thing.

After the handshake both parties can send application data, which in the case of HTTPS is standard HTTP data, with both encryption and authentication.

In a variant of the standard TLS handshake, the server may request that the client sends a certificate by sending a CertificateRequest to the client.

A graphical overview of the TLS handshake can be seen in figure 1.

2.2 TLS sessions

The TLS protocol also supports sessions by using a SessionID which can be sent during the handshake or along with application data. The support for sessions are only meant to increase performance and there are no security benefits from having session support in TLS [26].

\(^{1}\)http://tools.ietf.org/html/rfc2616
\(^{3}\)http://tools.ietf.org/html/rfc2246
2.3 Certificates

Certificates are central to TLS as they provide authentication to the protocol. X.509 certificates are most common but TLS also support OpenPGP certificates.

The X.509 certificates are either self-signed or signed by a Certificate Authority (CA). In order to properly verify the identity of the other party, the verifier need to either have the certificate hard-coded into the trust pool in the case of self-signed certificates, or in all other cases either trust the CA which signed the other party’s certificate, or a CA which signed the intermediate CA.

If the validation of the certificates fails it may indicate a man-in-the-middle attack. In this case the traffic can be both eavesdropped and modified by the attacker which completely removes all protections provided by HTTPS over plain HTTP.

3 Related work

Georgiev et al. [17] showed in 2012 that several frameworks for SSL either make it too easy to disable certificate verification, or even fail to offer proper verification by default. They show the importance of proper authentication when using encryption. If the communication is not authenticated properly using certificates, an attacker can easily remove the encryption without any party noticing the attack, resulting in the attacker getting access to sensitive data, such as passwords, and being able to modify the content of the traffic.

Later in 2012, Fahl et al. [14] performed a study where they analyzed the use of HTTPS by applications in the Android Market (now named Google Play Store). A total of 13,500 applications where examined using static code analysis. They found that 1,074 applications used incorrect HTTPS configurations which either accepts any certificate or any certificate signed by a trusted Certificate Authority (CA). They did not look at any known vulnerabilities to an otherwise secure SSL configurations but limited the study to incorrect use of HTTPS by application developers. They also limited the study to only the Android smartphone system.

Since 2012, when the paper was published, several attacks [3, 6, 25, 33] has been created by researchers which allow an attacker to partly or fully break the encryption of properly configured HTTPS communication channels. This paper will evaluate which applications use HTTPS communication vulnerable to these attacks.

Much work [7, 13, 29, 30] has been done on evaluating the security of the Android system. The focus, however, has mostly been on app isolation, permissions, and kernel protection. While this paper will only focus on communication security it is important to understand how the security of the smartphone platform was designed by Google and Apple. Network communication provides a large attack surface for gaining control over execution on the device. If an attacker is able to break the security of a communication channel, he can gain access to information and use it in a separate attack. Examining communication can also be done to detect the presence of a malicious or infected application.

3.1 Android security

The Android system uses an open approach to third party application distribution with its Google Play Store (previously Android Market). Since published applications are not manually reviewed, Google instead relies on security mechanisms both on the distribution servers and on the devices themselves. Enck et al. [13] provide an overview of the Android architecture from a security perspective. Each application runs as a separate user which provides process isolation due to the Discretionary Access Control (DAC) in Linux. Further, each application requests access to a number of permissions which are presented to the user during installation. The system checks these granted permissions when the application makes certain protected Application Programming Interface (API) calls.

While Shabtai et al. [30] state that Android security was designed with extensive care, they note that the permission system has several weaknesses. Users are not offered any fine grained control of what permissions to grant the applications. If an application requests access to permissions the user are not comfortable with granting, the only choice is to not install the application at all. They provide suggestions of improvement to the permission system, such as the ability to grant only a subset of requested permissions, and also advocate for the inclusion of Security Enhanced Linux (SELinux) to further protect the Linux kernel using Mandatory Access Control (MAC).

Using MAC to protect the kernel is a practical and useful measure against many types of attacks. As malware on Android continues to rise [29], rootkits such as one developed by You and Noh [37] can be used to compromise a device and steal sensitive information. Most such malware comes in the form of trojans [5] and Forristal [15] has provided a method to inject arbitrary code into application without invalidating their signature. This is done by duplicating a file and cross-reference it in or-
order to fool both the Java verifier and the Android verifier. Using this method Forristal was able to hijack the sandbox available only to system level applications.

Smaﬂey and Craig [31] incorporated SELinux in Android, enhancing the protection of the Linux kernel and prevented several attacks. This has later been incorporated into the ofﬁcial Android Open Source Project (AOSP).

Communication channels can be used to bypass the permissions system. Orthacker et al. [23] found a weakness where applications with different permissions could share data and thus forgo the permissions granted to a particular application. One application without access to a certain permission could pass the data to a second application which is granted the permission, allowing the data to be processed in a way not originally intended by the user. They suggest enforcing permissions on Interprocess Communication (IPC) channels to prevent this sort of attack.

A larger evaluation of security on Android was done by Berger et al. [7] where they found discrepancies between the documentation of Bluetooth communication available to developers and the actual implementation in the Android system itself.

While most previous research has been focused on code execution there has been several weaknesses identified regarding communication, both between applications and between an app and the Internet. Since Google does not use a manual review process it is important that the tools used to automatically review submitted applications can detect weak conﬁgurations and vulnerable communication channels.

### 3.2 iOS security

In contrast to the open model of Google’s application distribution, Apple uses a much stricter policy on its App Store. Each application is manually reviewed by Apple prior to being published on the App Store. This has however not stopped researchers from ﬁnding vulnerabilities in the iOS platform. In July 2007, just a month after the release of the ﬁrst iPhone, Schwartz reported in New York Times [28] about a vulnerability found by Charles Miller allowing him to gain full access to the device. Miller has since found several vulnerabilities in iOS allowing him to break out of the restrictions imposed by Apple [21].

Even though the restrictions put onto users and developers by Apple may have limited the spread of malware, it also served as motivation for the development of jailbreaking which has caused ﬁnancial loss for both Apple and its initial carrier partner, AT&T [24].

Wang et al. [33] demonstrated how to bypass the Apple review process and successfully upload a malicious application onto the App Store. They ﬁrst created an exploit which accessed protected system calls, bypassing the permission system of iOS and thus bypassing user interaction when performing sensitive operations. They then decomposed the exploit and injected it into various places of an otherwise legitimate application. Lasty they inserted a vulnerability which could be triggered remotely to assembly and run the exploit code.

While there has been less malware running on iOS than on Android, there is still no checks to ensure that applications on App Store use correct HTTPS conﬁgurations or even secure communication at all. To the knowledge of this author there has not been any survey of how common it is for applications on App Store to use weak communication channels.

### 3.3 SSL/TLS

Most communication on the web is secured through HTTPS which uses TLS to ensure both conﬁdentiality and authenticity. TLS however is based on a certificate scheme and there are many ways an app using HTTPS can be conﬁgured in a way that compromises this security.

A survey by Ristic [27] looked at invalid certiﬁcates and why they fail veriﬁcation. He found that most failures was due to the certiﬁcate having expired. The second most common reason for failure was that the certiﬁcate was self-signed. Self-signed certiﬁcates are not signed by a trusted CA but by the entity of the certiﬁcate itself. This means that unless the certificate is explicitly trusted it cannot be veriﬁed.

In 2010, Cheng et al. [9] demonstrated a man-in-the-middle (MITM) attack on all versions of SSL by exploiting a weakness during the initial handshake. This allows the attacker to retrieve the full plaintext of the communication.

Since HTTPS introduces an overhead on the communication and thus a performance penalty, many websites opt to only protect particularly sensitive pages with HTTPS. Herzberg [18] notes that a large portion of websites does not protect the initial fetching of login pages but only use HTTPS to submit the form over a secure channel. This allows an attacker to modify the login page before it arrives at the user.

The HTTPS scheme relies on trusted Certificate Authorities (CAs) to verify the certiﬁcates used when establishing a secure connection. Every CA is able
to sign any certificate for any domain. Dacosta et al. [10] suggest a solution to revelations that some CAs has given out certificates to third parties, allowing them to decrypt secure communications using those certificates. They suggest using Direct Validation of Certificates (DVCert) which uses cryptographic mechanisms to allow the user to directly validate a certificate without relying on an authority.

Xia and Brustoloni [36] made a study on how users behave when they are confronted with a warning about an invalid certificate. They found that most users simply dismiss the warning and those that are presented with an invalid certificate will only briefly examine it before deeming it acceptable. They do however find that users are more reluctant to send passwords over insecure channels. Sunshine et al. [32] found in a similar usability study that over 80% of users were willing to bypass an invalid certificate regardless of which website they were visiting.

A possible solution to weaknesses introduces by such user behavior is presented by Fung and Ch-eung [16]. They suggest a stricter enforcement of TLS certificate validation in JavaScript.

All these solutions aim at mitigating many of the possible weaknesses of a badly configured HTTPS communication channel. They stretch all the way from changing the way certificates are validated to aiding users in spotting signs of vulnerabilities in their communications. Most of these solutions have not been implemented in either iOS or Android, however. This means that many applications may use insecure communications even when using HTTPS and users may not even notice it.

3.3.1 Attacks against TLS

Even when HTTPS traffic is used properly by a developer and the certificates are verified as they should be, the traffic may still be vulnerable to attacks. In recent years there has been several attacks against TLS published by researchers in the field. These attacks are discussed below.

**TLS renegotiation.** In 2009 Ray and Dispensa [26] published a chosen-plaintext attack against TLS. The attack forces renegotiation in order to perform a man-in-the-middle attack. By leveraging the pipelining and keep-alive facilities in HTTP 1.1, the researchers were able to attach a custom request as a prefix onto a request sent by the client. By leaving out the final line break inside the attacker’s requests, the first line of the clients request will become part of the attacker’s header. In HTTP this first line is the resource being requested by the client.

TLS allows any party to request a renegotiation and the client can specify a session ID to immediately jump between different negotiated sessions. The result is the ability to carry out a chosen-plaintext attack against authenticated HTTPS communication, an attack which is used in other attacks in order to retrieve the plaintext of the communication. The attack can also be used to hijack victims HTTPS session.

This vulnerability was fixed in RFC 5746\(^6\) as an extension to TLS. A survey by Trustworthy Internet found\(^6\) that as of November 2013 a majority of public websites, 83.4%, had upgraded to only allow secure renegotiation.

**BEAST.** Another attack, known as BEAST (Browser Exploits Against SSL/TLS), by Duong and Rizzo [11] uses a previously known vulnerability in the cipher block chaining (CBC) present in TLS version 1.0. While the vulnerability was previously known there had not been any practical attacks using it until Duong and Rizzo demonstrated BEAST.

The BEAST attack needs an attack model where the attacker is both able to inject a prefix to a request from the victim as well as send a request at the end of the victim’s session. This can be done by redirecting the victim to custom JavaScript, Flash or a Java applet, by means of an iframe or similar.

While the attack is a chosen-plaintext attack, the full plaintext can be recovered after enough guesses. In practice, however, the attack will most likely only need to recover a specific part of the plaintext such as a session cookie.

**CRIME.** One year after publishing BEAST, Duong and Rizzo announced [12] CRIME (compression ratio info-leak made easy). It is an attack against both HTTPS with TLS compression, and the SPDY protocol. Similar to BEAST, CRIME is an oracle attack, but instead of using information leakage in CBC, CRIME uses information leakage in compression algorithms to infer whether a guess is correct or not.

The result is a chosen-plaintext attack where the attacker can uncover a chosen part of the plaintext (or the complete plaintext with enough guesses). This is done by injecting plaintext into the encrypted package and measuring the compression ratio compared to the size of the injected prefix. If the prefix contains text identical to some text

\(^5\)http://tools.ietf.org/html/rfc5746

\(^6\)https://www.trustworthyinternet.org/ssl-pulse/
in the victims request, the compression algorithm will compress both occurrences using references.

The injection of plaintext is done in the same way as in BEAST, namely by getting the victim to run custom JavaScript by injecting an iframe or redirecting an HTTP request. The JavaScript code makes a call to the third party server which will carry any session cookies the victim has previously established with the server. The attacker can then measure the size of the encrypted package and infer the compression rate.

**Lucky 13.** In February 2013, AlFardan and Paterson [3] published the Lucky 13 attack which uses information leakage from compression rate in or-

time

to carry out an oracle attack against

information leakage from compression rate in or-

time
time

time

time

time

as in BEAST, namely by getting the victim to run custom JavaScript by injecting an iframe or redirecting an HTTP request. The JavaScript code makes a call to the third party server which will carry any session cookies the victim has previously established with the server. The attacker can then measure the size of the encrypted package and infer the compression rate.

AlFardan and Paterson prove the contrary, that the timing channel will leak enough information for an attacker to be able to recover the plaintext using statistical analysis.

Since the Lucky 13 attack is a side channel attack, it is wholly dependent on the implementation and not the algorithm itself. As of the time this report was written most implementations has issued patches which fixes this vulnerability.

**RC4 biases.** A few months after publishing Lucky 13, AlFardan et al. [3] published an attack on the RC4 cipher in TLS. They noted that since there is a bias for certain values in the beginning of the keystream of RC4 an attacker can infer the plaintext if it is encrypted many times by different initial keys.

The improvement over Lucky 13 is that the attack against RC4 does not rely on measuring time or injecting plaintext into the victim’s session. It is however beneficial to the attacker if it is possible to force the victim to send the same encrypted plaintext over multiple TLS sessions. This may occur naturally but the attacker can force such behavior either by having the victim run custom JavaScript using a method similar to that in BEAST or CRIME, or by resetting TCP connections.

The attack works against all versions of TLS when the RC4 cipher is used.

**TIME.** In March 2013 Be’ery and Shulman [6] published an attack similar to CRIME which they named TIME. Similar to CRIME the attack uses information leakage from compression rate in order to carry out an oracle attack against TLS, but instead of measuring the length of the encrypted result, it uses timing to infer the compression rate of a guess.

The benefit of the TIME is that it removes the requirement to inject traffic into the victim’s session; the attacker only need to have eavesdropping capabilities. The downside is that timing attacks are sensitive to jitter and the quality of the communication channel.

Similar to CRIME, the TIME attack requires compression to be used in the TLS protocol.

**BREACH.** The latest attack against TLS which is profiled in this paper was published in July 2013 by Prado, Harris and Gluck [25]. Their attack, named Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH), is an improvement of the CRIME attack by Duong and Rizzo. Instead of using compression in the TLS protocol, BREACH measures the GZIP compression at the above layer, in HTTP.

Similar to CRIME, the BREACH attack requires an attack model where the attacker can both eavesdrop on the victim and inject traffic into the stream, for example by using JavaScript.

Since BREACH drops the requirement of compression in the TLS protocol and moves that requirement into the layer above, which is the encrypted payload, it is not possible to determine whether or not the traffic is vulnerable to this attack by simply looking at it. However, since compression in the HTTP data is almost ubiquitous due to the severe cost of disabling it, it can be assumed that all traffic is vulnerable to this attack.

**Vulnerable types of traffic.** Given the above list of known attacks against TLS, several types of traffic which is vulnerable to at least one attack can be identified.

First is all traffic using TLS version 1.0. This traffic can be decrypted by attaching either CBC or RC4, depending on which is used. If CBC-mode encryption is used, however, the attack would require a somewhat restricted attack model which may not be practical against smartphone applications, namely attaching both a prefix onto a request, as well as injecting a request into the established session. While this attack model introduces some limits it is does not make the attack impossible. Tactics such as attaching custom data into some other HTTP traffic and making session-carrying calls to the server can be used for attaching prefixes. By using keep-alive and HTTP pipelining, the second requirement can be met.

A second type of vulnerable traffic is that which uses either TLS with GZIP compression, or the SPDY protocol. This traffic is vulnerable to the CRIME
attack, which also requires an attack model where the attack can attach prefixes to requests. The attacker may also use the TIME attack which requires a different attack model: the attack must be near the victim but does not need to inject custom traffic.

Traffic vulnerable to renegotiation attacks, the BREACH, and the Lucky 13 attack are not identified in this paper. Results for the renegotiation attack is already available from Trustworthy Internet\(^7\), the BREACH attack depends on characteristics which cannot be determined without access to the plaintext, and the Lucky 13 attack is a side-channel attack, dependent on implementations. Also, the BREACH attack can be assumed to work in most cases due to the ubiquitous nature of HTTP compression.

### 3.4 Cloud security for mobile

Most communication an app running on a smartphone does is sending or receiving data to and from a cloud server. In addition to securing sensitive data by encrypting it and authenticating the server, it may also be important to make sure the server can verify the identity of the user using the app. Depending on the type of app and what kind of data the app or server need to process, many different security mechanisms may need to be used.

Smartphones put certain constrains on security mechanisms. A smartphone runs on a battery and must thus save as much power as possible. Cryptographic methods, however, are usually power consuming. Kamel et al. [19] use their earlier re-search product Architecture for electronic Docu-mEnt Plus their Transfers (ADEPT) to enhance the security of communication protocols in an efficient manner.

The synchronization system already present on Android is enhanced by Ai et al. [2] to allow each application on a smartphone to use a secure system for settings synchronization. Their system, named Application Settings Integration and Management Scheme (ASIMS), uses an open interface, which third party developers can utilize for synchronization, with a server that is verified by a trusted party such as Google.

Another enhancement of communication with the cloud is provided by Ahmad et al. [1]. They suggest authenticating with cloud servers using the (Universal) Subscriber Identity Module which is used to identify the phone with the carrier in telephony based communication.

### 4 Platforms

The tools used to develop third party software for handheld devices have become easier and more available than ever. Many developers today start out their career by building a smartphone application. Security, however, may often take a back seat while developing. It’s thus important that the tools and documentation helps the developer learn about the importance of security and follow established best practices.

The libraries were examined using the available documentation for the platforms, but their accuracy was not verified by an actual implementation. Berger et al. [7] found that there are discrepancies between documentation and actual implementation in Android, so the possibility that the documentation may not be fully accurate should be taken into account.

The amount of security related education provided by both the documentation and the code samples was determined subjectively by the author.

It should be noted that both the documentation and the code samples for both platforms reside online and may be subject to change. The version examined in this report was retrieved at October 2013.

#### 4.1 Documentation

The goal of a platform’s documentation is to both give an introduction to the tools available to the developer, as well as provide detailed information about the API inside the Software Development Kit (SDK).

The developers who are reading the documentation may come from different backgrounds with different amount of experience. Some might already be well informed about software security while others might not. If the documentation assumes some minimum level of prior knowledge it is desirable that the readers who are lacking the required skills are pointed to resources where they can acquire them before continuing to read the document.

It is also desired that the documentation not only informs about the security mechanisms that are provided by the platform, but also educates the reader on best practices and how to integrate security into the development early in the design process.

\(^7\)https://www.trustworthyinternet.org/ssl-pulse/
4.1.1 Android

The Android documentation\(^8\) is available online and is separated into three categories:

**Design**
Provides documentation on how to visually design Android applications according to the design language of the system.

**Develop**
Teaches developers how to build applications with code samples and library references, along with a training program.

**Distribute**
Explains the Google Play store, allowing developers to distribute and monetize from their applications.

This paper will only examine the **Develop** section and how it helps the developers build secure applications.

The network section of the training documentation explains how to communicate over HTTP using either the Android native class HttpURLConnection, or the HttpClient provided by Apache. While both support HTTPS, Google recommend using the former but without motivation.

There are no links to the security section from the network section. The single mention of HTTPS is the note that it’s supported by both aforementioned classes. The security section must thus be navigated to from the sidebar which limits the exposure. Only developers explicitly interested in developing secure applications will likely navigate to this page.

The security training section, seen in figure 2, begins with a **Security Tips** subsection which explains secure storage on the device, and the permissions system. Network security is divided up into Internet Protocol (IP) communication and telephone communication. The HTTPS protocol is mentioned and recommended over plain HTTP, with the argument that devices often connect to public Wi-Fi hotspots. Android provides SSLSocket for low level communication over sockets and HTTPSURLConnection for more high level secure communication.

After discussing the importance of input validation to prevent attackers from remotely exploit an application, the documentation goes into data privacy. The documentation only suggests that the developer handles sensitive data with care and not inadvertently expose it in plain text over network sockets. There is no mention how to best protect sensitive data.

The security section contains a subsection dedicated to cryptography where SSL is again mentioned. The text recommends using existing cryptographic systems instead of building custom system. The documentation recommends using SecureRandom to initialize cryptographic keys but fails to mention a recently discovered weakness [20].

---

\(^8\)http://developer.android.com/
in SecureRandom, leaving only applications using their own custom pseudorandom number generator able to generate secure keys.

The Android security documentation contains a section dedicated to HTTPS and SSL which begins by explaining the SSL/TLS protocols and digital certificates.

In a later subsection, possible problems regarding certificate verification are listed. The reader is taught how to solve these problems without compromising the security of either the device or the data.

The certificate validation issues listed are the following:

**Unknown CA**
Occurs when a certificate is signed by a CA which isn’t trusted by the system. Google recommends using TrustManager to add the specific CA as trusted by the system. Sample code to achieve this is provided. Google also warn against a practice it claims is recommended by third party websites where the developer creates a TrustManager which trusts every certificate.

**Self-signed certificate**
If the certificate is self-signed Google recommends the same practice as when the CA is unknown.

**Missing non-root CA**
If an intermediate CA cannot be reached to verify a certificate it is recommended to include the intermediary certificate on the server or treat the CA as unknown and follow the practices recommended in the first case.

Next the documentation lists common problems with verification of the hostname. Virtual hosting may cause issues with hostname verification as several different domain names may map to a single IP address. The solution Server Name Indication (SNI) is mentioned which is supported in TLS 1.0. HttpsURLConnection supports SNI but HttpClient does not which is one of the reasons why HttpsURLConnection was previously recommended.

A specific warning is given against providing a custom HostnameVerifier which can be used as an alternative method when dealing with virtual hosting. A code sample for a secure use of replacing the verifier for a URLConnection is provided.

An entire subsection is dedicated to warn about using SSLSocket since it does not come with certificate verification but leaves that up to the developer. A code sample is provided for establishing a connection over SSLSocket and verifying its certificate.

Certificate pinning (restricting the number of trusted CAs for a given connection) and blacklisting is mentioned but no code samples are provided demonstrating this technique. Instead developers must implement such features manually.

The Android documentation is well structured and contains many code samples. There are a few references to security from the networking section and the documentation does not warn readers about recently discovered vulnerabilities in the APIs.

### 4.1.2 iOS

Similarly to the Android documentation, the iOS documentation is also provided in form of a website. It contains a large set of documents and the website allows the developer to view a subset of the documents using a filter. The subset can be all documents of a certain type (Article, Tutorial, Release note, etc.) or belonging to a certain topic (Graphics & Animation, Networking & Internet, Security, etc.). The documents can also be filtered depending on which of the different iOS frameworks they belong to. There is also a searchbox allowing a reader to quickly locating a specific document or set of documents using keywords. A screenshot of the Security topic filter can be seen in Figure 3.

The iOS documentation for developers contains a total of 1,700 documents. There are 121 documents under the Networking & Internet topic and 19 documents under the Security topic. The Security framework inside the Core OS Layer contains 13 documents and the CFNetwork framework inside the Core Services Layer contains 17 documents. The documents are always presented in a list.

Under the Getting Started section, the document Network & Internet Starting Point can be found. This document starts by suggesting that the reader first read through the document named Document Transfer Strategies which lays out an overview of how to design an application which uses the network to transfer data to and from a server. The document specifically points out security related issues such as malicious attacks and authentication. It also has an entire section devoted to TLS.

---

Network & Internet Starting Point continues to explain that network communication in iOS can be done on three different abstract levels. Highest is the use of `URLs` using the `CFURL` and `NSURL` classes (the `CF` prefix indicates a C-based API while the `NS` prefix is for APIs based on Objective-C) which allow the developer to establish connections to servers using a URL and download the response.

If a connection is needed for a longer session the developer is provided with `CFHTTPStream`. There is no mention of `HTTPS` but the developer is referred to the `CFNetwork Programming Guide` which further explains the `CFHTTPStream` class and how to use `HTTPS` for secure communication.

A more low-level method for using network communication in iOS is Sockets. iOS provides access to the sockets using the `bsd socket(2)` API from UNIX.

The `Network Overview` document is not prominently featured. It can only be found under the topic `Networking & Internet` or under the `Guides` section where it is fairly far down the list of documents. The same goes for the `Networking Concepts` document.

At the end of the first section the `Network Overview` document instructs the developer to:

> Design your software carefully to minimize security risks and to correctly take advantage of security technologies that are available, such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS).

The document contains an entire section dedicated to HTTPS and another section discussing secure networking. The reader is educated on four different forms of attacks: snooping, man-in-the-middle, injection and buffer overflows. The document instructs on how to provide secure communication using `URLs`, streams and sockets.

While `NSURLRequest` will use HTTPS automatically when the URL is prefixed with `https://`, using TLS with streams require the developer to call `setProperty:forKey:` on the object and specify `NSStreamSocketSecurityLevelNegotiatedSSL` as the property and `NSStreamSocketSecurityLevelKey` as the key.

The developer is discouraged from using sockets for secure communication since it requires that the developer implements the encryption and decryption themselves, either by using the Secure Transport API or a third party library such as OpenSSL.

The reader is also introduced to additional security-related protocols available in iOS such as Kerberos and `SSH`. The information is very brief and only aims at informing that support for the protocols exists.

The document lists a number of common mistakes developers may make when attempting to implement secure communication. It is pointed out that both the data and the second party should not always be completely trusted. The receiver or sender may be impersonating someone else and the data may come from an untrusted source. The reader is referred to two sections in the `Secure Coding Guide` for further information on how to protect against potential attacks.
The reader is then educated about the dangers of improperly installed certificates and contains a strong warning against disabling certificate chain validation. Instructions on how to properly deal with self-signed certificate are provided but there is no sample code.

The Security Overview document encourages the developer to protect sensitive data, be careful about untrusted data, and authenticate the participants of secure communication. It also urges the developer to not reinvent the wheel as inexperienced developers may create a faulty implementation of an otherwise secure protocol, or may make mistakes when trying to design a secure protocol themselves.

The document Override TLS Chain Validation Correctly aides the developer in implementing their own verification of TLS certificates in a secure manner. The text discusses when such measures are needed and when to avoid doing so. It contains several code samples and assumes prior knowledge of cryptography from the reader.

The iOS documentation is composed of a large corpus of documents which can be hard to grasp. The security related documents are usually hidden far down the long list of documents but they contain detailed discussions on best practices and often point the reader to further information. While there are some code samples provided the reader is often left to manually construct the ideas and tips into code.

4.2 Libraries

While the documentation can be used to teach developers how to follow best practice regarding security, they also teach the developer about the APIs available on the platform and how they work. The libraries exposing these APIs range from low-level mechanisms, which allow developers to work directly with network packets, to more higher-level abstractions, which allows for quick access to resources on the network.

This section lays out the most common methods of implementing network communication on both platforms and briefly discuss how they work and differ from each other.

4.2.1 Android

In section 4.1.1 we learned that developers on Android have two main classes available to them for implementing secure network communication, namely HttpURLConnection and SSLSocket. The former is more high-level, the developer doesn’t have to implement the handshake procedure, maintain sessions, or validate certificates. The latter, on the other hand, only provides the developer with a socket where raw packets can be sent and retrieved.

In addition to these two classes there is also HttpClient from Apache, but this classes does not provide SNI which is why Google suggest that developers use HttpURLConnection instead.

A developer using SSLSocket must manually implement verification of the server’s certificate. This can however be done with relative ease by retrieving the default HostnameVerifier used by the HttpsURLConnection class. The developer then only needs to call verify on the class in order to verify the certificate properly.

In the case of a self-signed certificate, the developer must manually create a TrustManager which trusts the certificate and use this for the connection. This is done in several steps. First the developer must generate a Certificate from the certificate file. Then a KeyStore must be created which trusts the signer of the certificate. The custom TrustManager must then be made use of this KeyStore, and then an SSLContext can be created from the custom TrustManager. Lastly a SocketFactory can be retrieved from the SSLContext. This SocketFactory can then be used whenever a secure connection need to be made to any server using the certificate.

As noted in section 4.1.1, the documentation contains a complete code snippet for implementing the above steps.

4.2.2 iOS

On iOS, developers have more options available when creating secure network communication. There are three main abstraction levels available to developers. Highest up is downloading a URL using a simple GET request. Lower down is using streams to issue more complex HTTP requests such as POST. The most low-level of interacting with the network is by using the socket(2) API from BSD.

If the developer does not have a certificate signed by a trusted CA he needs to implement a custom trust for a specific certificate. This is done by adding the certificate to the KeyChain before issuing any HTTPS requests. The procedure is similar to that on Android. First the developer creates a SecCertificateRef object from a certificate file. Then an NSDictionay object is created containing kSecClass = kSecClassCertificate, and kSecValueRef = the certificate object. This dictionary is then added to the KeyChain using the SecItemAdd method.
The code for adding a custom certificate to the KeyChain is less than ten lines but it is not included in the iOS documentation. There are however code snippets available on third party websites, and the code sample named AdvancedURLConnections discussed in section 4.3.2 contains code for achieving this among other things.

4.3 Sample code

The Software Development Kits (SDKs) for both Android and iOS provide the developer with a large set of tools such as libraries, an Integrated Development Environment (IDE), smartphone emulator, and sample code.

This section looks at the provided code samples related to network and security.

4.3.1 Android

The Android SDK comes with a special tool for downloading sample code. There are a total of 69 sample projects of which 49 are legacy projects. Only the non-legacy projects have been examined. The SDK version used is 4.3.

The sample projects are categorized using folders. There is a category named security which indicates a special attention to security from Google. The category however contain only a single project which exemplifies the use of the Android key chain for storing keys and passwords.

The connectivity folder contains three sample projects.

BasicNetworkDemo shows how to check the connectivity of the device, if it is connected and if it’s via Wi-Fi or cellular. No data is transmitted and there is no mention of security.

BasicSyncAdapter shows how to download an eXtended Markup Language (XML) feed and use it to synchronize the state of the application. The project uses HTTP to achieve this and does not mention HTTPS explicitly or even security. The project does however work with URLs prefixed with https://.

NetworkConnect downloads data over HTTP and similar to the BasicSyncAdapter the project does support URLs prefixed with https:// but does not mention either HTTPS in particular or security in general.

The Android code samples are very few and does not touch on security at all. They do not mention how to integrate security into the code early in the process and fails to mention the HTTPS protocol at all. The security folder contains only a single project.

The code samples need to be sought out explicitly and can only be obtained via a separate download manager.

4.3.2 iOS

The sample code for iOS are available from the same website as the documentation. The documents of the Sample Code type contain a brief explanation of the project and a link for downloading the project.

There are a total of 155 sample projects available with no further categorization. Instead the search tool provides the ability to filter the list of samples by keywords. This works just as well since it’s possible to search for a specific topic such as Networking & Internet.

There are two samples under the Security topic, but only one deals with communication security:

CryptoExercise uses the cryptographic API of iOS to create a web services which uses symmetric key encryption and message digests.

Under the Networking & Internet topic there are nine projects:

CustomHTTPProtocol extends the NSURLProtocol used by a UIWebView object. This allows it to intercept HTTPS traffic from the web view and implement a custom authentication mechanism. This particular example implements the verification of a self-signed certificate from the server.

SimpleFTPSample uses the CFFTPStream and NSURLConnection to establish an FTP connection. It does not support authentication or any security features. It expressly focuses on network and not security.

WiTap establishes a peer-to-peer connection or Wi-Fi. It does not touch on any security aspects of networking.

SimpleNetworkStreams creates a simple server-client connection using the NSStream API. It does not contain any security mechanisms but comes with instructions on the mechanisms needed to turn the sample into code suitable for production.

SimpleURLConnections uses simple HTTP to download and upload data. It is however possible to add support for HTTPS using the built in APIs in iOS.
AdvancedURLConnections shows how to implement custom certification verification in order to verify self-signed certificates. It also provides client identity. The instructions for how to configure the server comes with a warning that the configuration may be insecure.

MVCNetworking uses the Model-View-Controller design pattern to create an application which downloads images from a web server and displays them in a gallery. It supports both HTTP and HTTPS.

Reachability shows how to detect when the device is connected and which types of connectivity are available. The only transmission of data is to an HTTP server to determine reachability.

BonjourWeb implements a browser for showing services detected on the network using Bonjour and the NSNetService API.

The iOS code samples aiming at showing how to do basic communication do mention security but do not implement secure communication themselves. There are two code samples which explicitly show how to use cryptographic API calls to implement secure authentication and certificate verification.

The code samples are easy to download and are available from the same lists as the other documents in the iOS documentation.

4.4 TLS support

As was be shown in section 3.3.1, the security of a properly validated HTTPS session very much depend on which version of TLS and cipher suit is agreed upon by the client and server. During the TLS handshake the client and server will agree on the highest version of TLS supported by both parties as well as the strongest, mutually supported cipher suit. For this reason it is important that the built-in libraries of each platform supports new and secure versions of TLS as well as strong cipher suits.

Android 4.1 introduced an upgrade to JSSE that supports TLS version 1.2. iOS introduced support for TLS 1.2 and 1.1 in version 5.0.

Both Android and iOS support the RC4 cipher and AES under CBC encryption mode. iOS also supports GCM and CCM encryption mode for AES.

No platform support compression in the TLS protocol, but both platforms support the extension for secure renegotiation.

Both platforms also support third party libraries such as CyaSSL, NSSSL, and PolarSSL. These let the developer use configurations of TLS not natively supported by the platform.

4.5 Summary

The tools available for developers on Android and iOS allows for quick and easy communication over the HTTPS protocol with proper certificate validation. While it is possible to implement custom certificate validation on both platforms, they both recommend developers to do so with caution since it may introduce critical bugs weakening security.

Self-signed certificates is a common reason to implement custom certificate validation and both platforms mention solutions for this problem. Google provides code for doing this directly inside the documentation, while Apple has a downloadable code sample which provides this among other things.

The code samples provided by Apple are more educating on security than those provided by Google. The code samples for iOS are also easier to access since Google distributes all code samples via a dedicated download manager, in contrast to Apple which places code samples on the same website as the online documentation.

Both platforms has introduced support for TLS version 1.2 in their latest versions but the developer must specifically target these versions in order to take advantage of the support for newer versions of TLS as well as the strongest cipher suites.

5 Applications

While the documentation and code samples might guide developers in making secure applications, developers can still choose to cut corners and develop applications which are insecure.

In this paper, applications are examined by using both static code analysis and manual analysis of their communication. Only applications on Google Play Store, however, are analyzed by reverse engineering due to time constraints. Tools for decompilation and code analysis are more widely available for Android applications. A similar analysis on iOS, while possible, would require more effort.

During static code analysis it is determined which classes and methods a certain app contains and what those methods return. This allows for determining if there are any certificate validators inside the code of the application which will validate certificate that are invalid.

The most downloaded applications are then ex-
amed manually by analyzing their communication. By inspecting the version of the protocols used, if the traffic is compressed, and the cipher suite used, it is possible to determine if the traffic flow is vulnerable to any known attacks.

5.1 Selection

More downloaded applications pose a larger threat if they are found vulnerable, as it will affect a larger user base. For this reason applications with a large number of downloads are more interesting to examine.

Both the App Store and the Google Play Store separate applications into categories. Apple’s App Store contains the following categories:

- Books
- Business
- Education
- Entertainment
- Finance
- Food & Drinks
- Games
- Healthcare & Fitness
- Lifestyle
- Medical
- Music
- Navigation
- News
- Photography
- Productivity
- Reference
- Social Networking
- Sports
- Travel
- Utilities
- Weather

Google’s Play Store on Android separates applications into the following categories:

- Books & References
- Business
- Comics
- Communication
- Education
- Entertainment
- Finance
- Games
- Health & Fitness
- Libraries & Demo
- Lifestyle
- Live Wallpaper
- Media & Video
- Medical
- Music & Audio
- News & Magazines
- Personalization
- Photography
- Productivity
- Shopping
- Social
- Sports
- Tools
- Transportation
- Travel & Local
- Weather
- Widgets

Under each category there are usually several subcategories: popular, new and top grossing, which are then split into free and paid.

In order to maintain an even spread of the sample and also allow for comparison between categories, it is desired to select an equal number of applications from each category. However, some categories contain very few applications.

The automatic nature of the static code analysis on Android applications allows for every app listed under every category to be examined. This gives a theoretical maximum of 67,500 applications since the store allows for retrieving 500 applications under each of the five subcategories under all 27 categories. However, as previously stated, some categories do not contain a full set of 500 applications. There’s also a lot of overlap both between categories and between subcategories.

For manual traffic analysis 150 applications were selected from each platform, all of which belonged to the top 15 most downloaded in their respective category. It should be noted that the resulting data will thus be biased toward popular applications. Developers of highly popular applications are likely to have more resources than the average developer.

5.2 Test setup

The following section will detail the tools used to carry out the analysis as well as how the various components are connected to each other. It describes both the custom script developed during this research which is used to reverse engineer the Android applications on Google Play Store. The network setup for intercepting the applications HTTPS traffic is also explained in this section.
5.2.1 Code analysis

The static code analysis on Android is done in three steps. First the Google Play Store is browsed and a list of applications is compiled, then the applications are downloaded one by one and analyzed, and lastly the results are compiled into graphs and tables. All this is done by a script, named hermes\textsuperscript{11}, created as part of this research.

hermes uses several third party tools for carrying out these tasks. To access the Google Play Store the script uses Google Play Unofficial Python API\textsuperscript{12} which is a Python wrapper around an unofficial API for the Google Play Store. This API allows for browsing categories, listing applications inside categories, downloading applications, and fetching meta data.

For static code analysis the mallodroid\textsuperscript{13} software written by Fahl et al. [14] is used. It is a component used inside the androguard\textsuperscript{14} framework which provides general reverse engineering capabilities of APK (application package file, the file format for applications on Android) files.

When the Google Play Store is browsed the script also fetches meta data for each application. This meta data includes the author of the application, when the application was last updated, the rating of the application, the number of times the application has been downloaded, whether or not the application requires the Internet permission, the price of the application, and the category the application belongs to. The script only lists the first 500 applications in each category, it does not perform any searches.

After compiling a list of the applications in each category, hermes starts to perform a deeper analysis on the free applications that require the Internet permission. These applications are downloaded one by one and analyzed using the mallodroid component. The result is then saved as additional meta data of the application.

During the static code analysis certain pieces of code are searched for in order to identify applications containing insecure code.

Custom TrustManager. Implementing a custom TrustManager class must not be insecure. In fact, it is needed for handling self-signed certificates or situations were the signing CA is unknown to the platform. If, however, the class method checkServerTrusted always returns either true or void, and throws no exception, it is highly insecure as this will accept any certificate without proper validation.

Custom HostnameVerifier. Similarly to the TrustManager, a custom HostnameVerifier must not by itself be insecure, it also is sometimes necessary, but when the verify method always returns true or void it is indeed insecure. Another type of insecurity arises when the verify method instantiates an AllowAllHostnameVerifier object.

\textsuperscript{11}https://github.com/ephracis/hermes
\textsuperscript{12}https://github.com/egirault/googleplay-api
\textsuperscript{13}https://github.com/sfahl/mallodroid
\textsuperscript{14}https://code.google.com/p/androguard
Insecure SSLCertificateSocketFactory. Applications are also scanned for code which calls the static GetInsecure method of the SSLCertificateSocketFactory class as this will return a SSLCertificateFactory with all the verification checks disabled.

5.2.2 Traffic analysis

Morrissey et al. [22] suggest a setup for intercepting and analyzing traffic. Their setup includes a router and an access point both connected to a computer via a network hub. This setup is overly complex for an experiment such as this, instead a more simple setup is used. In order to intercept the encrypted HTTPS traffic a proxy is placed between the smartphone and the Internet.

The proxy mitmproxy\textsuperscript{15} provides interception of both HTTP and HTTPS traffic in a transparent mode where the user does not have to make any explicit configuration in order for the traffic to be redirected through the proxy.

In this experiment a 2013 MacBook Air running OS X version 10.9 is connected to the Internet via an Ethernet cable connected using a USB dongle. The Internet connection is then shared using the laptop’s wireless card. mitmproxy is running on port 8080 on the local interface (with IP address 127.0.0.1) and the built in firewall Packet Filter (PF, originating from OpenBSD) is configured to redirect all traffic destined to either port 80 (HTTP traffic) or port 443 (HTTPS traffic) to 127.0.0.1:8080.

\textsuperscript{15}http://mitmproxy.org/

mitmproxy will by default replace all certificate with it’s own custom certificate, which is generated during run time. It connects to the destination server and retrieve information on the certificate prior to this generation which makes the certificate look like it comes from the intended destination. The generated certificate is signed by a custom CA certificate which is shipped with mitmproxy. Unless this CA certificate is installed into the client the generated certificates for all intercepted HTTPS traffic should fail validation.

mitmproxy can also be configured to provide a custom certificate. This allows for testing with a certificate bought from a CA and thus properly signed. The issue will instead be a mismatching hostname.

Exceptions can be added to the PF rules. This allows for letting some traffic, for example traffic destined to Facebook or Google, to be passed without modification while all other traffic is captured and analyzed. This is useful since some applications may send traffic to third parties during authentication. This authentication is often done by using code provided by the third party, which mean that the authentication traffic’s certificates may be properly validated while all other traffic may not.

This setup allows three kind of tests to be carried out: presenting the application with a certificate signed by an untrusted CA, or a certificate with a mismatching hostname, or leaving the traffic unaltered.

The first two tests are used to verify the results from the code analysis where improper verification
code is identified inside the application’s binary file. The third test is used to profile the HTTPS traffic against known attacks.

5.3 Results

During the static code analysis a total of 38,229 applications where found via the Google Play Store API. Of these applications a total of 33,101 applications (86.59%) required access to network communication. 27,807 were available free of charge. During this research only free applications which required the Internet permission were considered. This resulted in 23,932 applications being downloaded and analyzed using reverse engineering.

A total of 300 applications were further tested using manual traffic analysis, 150 on Android and 150 on iOS. The applications are all among the 15 most popular free applications in their respective category.

This gives a slight bias of the result toward popular applications. This could skew some of the numbers since developers behind highly popular applications are likely to have more resources than other developers.

It is also worth noting that applications tested during manual testing will be counted twice if they appear on both Android and iOS. In some cases, though, applications differed between the platforms. For this reason the applications are considered different for each platform in the aggregated results.

This section will present the results from both these tests. Firstly, applications’ tendency to offer basic protection of traffic data is examined. Secondly, the certificate verifications is looked at in applications which use HTTPS for communication. Thirdly, in applications using HTTPS, the traffic is profiled against the attacks discussed in section 3.3.1.

Lastly, the security of applications’ communication is correlated to their meta data in order to find any indicators of secure applications. Such correlations can aid users in choosing applications which properly protect their personal data.

5.3.1 Traffic protection

The first step in protecting traffic data is to encrypt the data. Naturally, if the data is encrypted it must also be authenticated. Otherwise a man-in-the-middle attack can easily be mounted and remove the encryption.

It is clear from the test results that HTTPS is the most common method of protecting traffic data. 48.33% of the tested applications used HTTPS to protect their communication traffic.

![Figure 6: Types of traffic observed.](image)

A breakdown of this data can be seen in figure 6. Only traffic to and from the developer’s own server is counted. Traffic not counted include tracking using third parties, advertisement, and static resources distributed using third party Content Delivery Networks (CDNs).

The difference between platforms regarding the share of unprotected data was small and within the margin of error for a 95% confidence level. The \( n-1 \chi^2 \) test shows that there is a 88% chance of applications on Android being more likely than their iOS counterparts to protect traffic by using HTTPS when accessing the network.

One might not consider HTTPS necessary when the traffic data is not sensitive enough to warrant protection. To see if this is the reason why applications might choose to use HTTP instead, the payload data of HTTP traffic is examined.

Sensitive data was observed in some of the unprotected traffic. Passwords, session cookies, and search queries for drugs, are some examples of sensitive traffic that was sent without any protection.

Around a tenth of all applications sending data unprotected included passwords in their traffic. Due to the small sample size it is difficult to give an exact estimate of how common this is. An adjusted Wald interval for 95% confidence can be used to give an indication, however. Such an interval shows that between 6.54% and 18.89% of all applications using HTTP will include user’s passwords in their traffic.

There is a 61% chance that applications using HTTP on Android are more likely than those on iOS to expose passwords.
The situation on Android and iOS regarding network security is better than that on the web where only 24.6% of the most popular websites use HTTPS according to a survey by Trustworthy Internet\textsuperscript{16} from November 2013.

There is still room for improvement, there are still applications sending sensitive data without any traffic protections. More usage of HTTPS, or some other protection, is desired.

5.3.2 Certificate verification

When using HTTPS for protecting traffic it is important to properly authenticate the party answering the request. If this is not done, the traffic can easily be compromised without the client noticing it.

For this reason it is important that applications not only use HTTPS, but also authenticate the server’s certificate properly. As noted in section 4.2.1, developers on Android can either use the native verifier that come built into the Android platform, or create their own verifier by extending either TrustManager or HostnameVerifier. Using a custom verifier is necessary when the certificate is either signed by a CA not trusted by the operating system, or when the certificate is self-signed.

An application containing either a custom TrustManager or a custom HostnameVerifier is classified as containing a custom verifier. If the custom verifier in addition has an empty checkServerTrusted or verify method respectively, it is considered a naive verifier. Lastly, code such as AllowAllHostnameVerifier or SSLCertificateSocketFactory->getInsecure() is classified as “bad”.

A full breakdown using this classification among applications requiring the Internet permission is shown in figure 7. It shows that 37.30% contain a custom verifier. Of these, 90.52% are naive.

While 29.06% of all free applications contain insecure code it is worth noting that the presence of insecure code does not necessarily make the application insecure. The code may be used only in specific scenarios or not used at all. This becomes apparent when looking at the results from the manual traffic analysis where only 4.05% of all applications tested on Android, 7.89% of those using HTTPS, fail to verify certificates.

An estimate of how likely an application containing insecure code is to actually use that code, can be made by combining the data from the static code analysis with the data from the manual traffic analysis. An adjusted Wald interval shows that, with 96% confidence, between 5.61% and 25.55% of all applications containing insecure code will also use it in production.

![Figure 7: Types of verifiers observed.](image)

It is worth noting that the data from the manual traffic analysis is biased toward popular applications. Since developers behind those applications are likely to have more resources available than the average application developer, this number may in reality be a bit higher.

On iOS there is no static code analysis to detect the presence of insecure code, but the manual traffic analysis shows that 3.33% of the applications, 7.46% of those using HTTPS, fail to verify certificates. Here there is only a 4% chance that there is a difference between the platforms, with applications on Android being slightly more likely than those on iOS to skip verification when using HTTPS.

Similarly to sensitive data sent over plain HTTP, there are occurrences of both passwords and search queries being exposed over HTTPS traffic where there is no proper verification of the certificates. The proportion of applications using insecure HTTPS that expose passwords lies between 42.89% and 90.80% with a 95% confidence.

Looking at passwords being sent over both plain HTTP and insecure HTTPS, between 4.31% and 10.12% of all applications on either platform expose users’ passwords with a 95% confidence. During the test an equal number of applications leaked passwords on both platforms.

In summary, certificate verification on both Android and iOS can be said to be mostly secure, but around one in twenty applications using HTTPS do not verify certificates properly. A large part of these also include sensitive data such as passwords in their traffic, which could be explained by a false assumption that merely encryption, without

\textsuperscript{16}https://www.trustworthyinternet.org/ssl-pulse/
proper authentication, is sufficient to protect sensitive information.

Such an assumption is naturally not true and the tests in this paper has shown that it is fairly easy to remove encryption from unauthenticated HTTPS traffic and expose sensitive information such as passwords and session data.

5.3.3 Vulnerable TLS traffic

While using HTTPS to protect traffic, and properly verifying the certificates to keep that protection intact, is a good first step toward securing the data being communicated, it is also important to use a proper configuration of cipher suites, compression, and protocol versions.

In section 3.3.1 a number of attacks against HTTPS were laid out. This showed that the ideal configuration is TLS version 1.2, no compression, and a cipher which does not use CBC mode encryption. The RC4 cipher should also be avoided.

The traffic seen from the manual traffic analysis shows that nearly every application using HTTPS is, in theory, vulnerable. Most notably is the use of TLS version. Every application use TLS version 1.0 instead of the latest version 1.2. The exception being the built in, and some third party on Android, web browsers which support version 1.2. On Android two very popular browsers were found to only support TLS version 1.0.

While TLS version 1.0 does not automatically mean that an application is vulnerable to an attack, it does point out that there is a theoretical vulnerability which may be exploitable. Some of the attacks in section 3.3.1 require specific attack models in order to work that may not be applicable in the case of smartphone applications.

During the manual traffic analysis, traffic from other parties were observed such as advertisement networks, user or device tracking, and CDNs for static resources. Most of this used TLS version 1.0 but one CDN provider used TLS version 1.2. Some authentication with third parties such as Facebook also used TLS version 1.2.

There was also a highly popular analytics provider which used SSL version 3.0, the predecessor to TLS, which is considered highly insecure.

Tests showed that 9.09% of applications with HTTPS traffic on Android used AES in CBC mode, while all other used the RC4 cipher. On iOS every application used AES in CBC mode.

This means that both the BEAST attack and the RC4 attack are possible to carry out in theory against applications on Android and iOS. The BEAST attack may not be practical as it requires a very specific attack model and the packets may be split by the client so to prevent the attack from working.

The attack against RC4 biases on the other hand, while being more effective if the attacker can repeat traffic, works under a much less restrictive attack model. It can very likely be carried out against smartphone applications using the RC4 cipher given enough traffic.

Looking at third party traffic, all used either RC4, or AES in CBC mode on Android, and all used AES in CBC mode on iOS. The one exception was some applications on Android using third party authentication, for example Facebook. This traffic used AES in GCM mode, and TLS version 1.2. A possible explanation could be that the traffic is being sent by an embedded web view based on the webkit engine, which supports a more modern version of TLS than that of the platform libraries.

No traffic used TLS compression which means that neither the CRIME nor the TIME attack are possible.

This paper does not attempt to verify whether the attacks are possible to carry out or not, either as-is or with some modification. Instead it only points out the theoretical vulnerability in the HTTPS traffic. The attacks are only practical under the proper attack model and with certain conditions.

5.3.4 Indication of security

Many users will most likely want to take steps to protect their personal data. Doing manual analysis of either an application’s binary file or its network traffic is, however, not a task most users will be likely to take.

If a correlation between the security of an application’s traffic, and some property of the application, can be found, it can be used to get a hint of how secure data used by that application will be.

Since the data from the traffic analysis is too small, the data from the static code analysis is used instead. Even though this data is limited to Android and only applications which contain insecure code, not application which expose sensitive data when actually used, it can still give useful indications.

Looking at the number of downloads, it becomes apparent that less downloaded applications are more likely to contain insecure code. This is illustrated in figure 8. Applications with 100 downloads or less have the highest share of insecure code. Highly downloaded applications also seem
to have a higher ratio of insecure code. While the number of applications with more than a hundred million downloads is very few, there is a 67% chance that a larger share of these applications contain insecure code than those with 1,000,000-100,000,000 downloads.

Another correlation found, though not useful as an indicator for a regular user, is that in categories with a high proportion of applications requiring the Internet permission, there is also a higher share of applications containing insecure code. This correlation can be seen in the graphs in appendix A.

Unfortunately, since the static code analysis did not identify the type of protocol used by applications, there is not enough data to spot correlations between rating or number of downloads, and usage of HTTPS over plain HTTP.

5.4 Summary

The resulting data shows that there’s a relatively large proportion of applications using HTTPS when compared to the number found on the web. There are however still quite a few applications which send sensitive data without using any protection of the communication traffic, thus exposing passwords to eavesdroppers.

When looking at applications that do use HTTPS to secure their communication, a large part of the applications contain code that do not properly validate certificates. Fortunately, this code seems to be mostly unused as in real life testing only a small part of the application actually accepted a faulty certificate.

There is however a non-insignificant share of applications that do accept faulty certificate and will thus be vulnerable to basic man-in-the-middle attacks. Indicators such as popularity and rating can be used to stay away from these applications as the trend seems to be that applications with at least 100 downloads and a high rating are less likely to contain insecure code.

No application during this test was found to use the latest version of the TLS protocol and all used either the RC4 cipher, or AES in CBC mode, thus exposing their traffic to known vulnerabilities.

There are attacks against these vulnerabilities that work against web browsers but, at least to this author’s knowledge, there has been no practical attack demonstrated against smartphone applications, where the situation is slightly different. The RC4 attack, though, is very likely to be practical against a majority of the smartphone applications using HTTPS, if there is enough repeated traffic available to the attacker.
There are a number of web browsers found on Android which do not support the latest version of TLS, thus being vulnerable to some of the attacks discussed in section 3.3.1. The situation here is different due to the nature of a web browser, making the attacks more likely to succeed.

6 User behavior

The last link in the chain of secure communication is the user. None of the tested applications, that properly verified certificates, allowed the user to continue using a communication channel when the certificate failed to validate. However, most browsers on both platforms do so.

Many websites employ the HTTP Strict Transport Security (HSTS) mechanism\(^\text{17}\). If HSTS is used then the browser will not show a button for bypassing the certificate warning if the certificate is signed by a trusted CA, but for another hostname.

If users are inclined to bypass certificate warnings when given the chance, this would introduce a way for attackers to carry out man-in-the-middle attacks even if the web browser only employs secure configurations of TLS and proper certificate verification.

Xia and Brustoloni [36], and Sunshine et al. [32] studied this behavior of users as discussed in section 3. These studies were both conducted in a lab environment. Users may react differently when they encounter such errors in their daily lives. Their behavior may also differ when using a personal smartphone or laptop instead of a lab computer.

In order to determine how users react when they are given the chance to accept a faulty certificate while in a natural setting, an experiment was made at the central train station in Uppsala, Sweden during a Friday.

In order to preserve the privacy of the users no data at all was saved after the experiment. As soon as the data was analyzed it was purged from the computer. The result from the analysis contained no personally identifiable information.

The setup used is similar to that describe in section 5.2.2 and can be seen in figure 10. A smartphone is tethered to a computer running the mitmproxy software, and the computer is sharing the Internet connection via the wireless network card.

6.1 Connecting to open network

Both platforms show indicators when a network is open, i.e. no encryption takes place and all traffic is sent in the clear. Connecting to such a network should therefore be done with caution and no personal information that is not encrypted in the application itself should be sent. Unfortunately, users have no easy way of knowing which applications protect traffic and which do not.

For this reason users should be wary of connecting to any open network as it will increase the risk of their personal data being exposed. As shown in section 5 there are several applications that do send passwords unprotected. There were also observa-

\(^{17}\)http://tools.ietf.org/html/rfc6797
tions of applications sending session data without protection, thus allowing an attacker to hijack the user’s session.

The SSID used for the experiment is the same one used by the free wireless network provided by the local university. This network has no password protection and instead uses a web-based login form for authenticating students when they attempt to access the Internet. Of course, during this experiment no such login form was presented to the users.

The result was that devices which had this network saved automatically connected to the laptop without needing user interaction. A test on the university also showed that if the laptop is close to a device, so to make the signal from the laptop stronger than the signal from the university’s access point, devices would also automatically send their data to the laptop and thus allow for man-in-the-middle attacks.

In order to get an estimate of how successful this technique was in getting users to connect a tool called device_sniffer18 was developed. This tool scans the surrounding area for 802.11 packages of types request-to-send, data, probe-request, and probe-response.

Special addresses such as the broadcast address and those used in spanning trees are filtered out. The output from device_sniffer shows the number of devices present that have their wireless interface enabled but are not already connected to a wireless network.

![Figure 11: Devices seen during the experiment.](image)

During the experiment at the train station, which lasted for two hours, a total of 622 devices sent traffic over the air and 45 of those devices did connect to the rogue hotspot. Nine of the connected devices were seen sending traffic which matched that of a web browser. An illustration of this can be seen in figure 11.

The large part of connected devices are most likely equipment such as computers and servers part of the station’s infrastructure.

6.2 Confrontation with invalid certificate

To measure the number of times security warnings regarding unverified certificates are being discarded by users, two types of data are collected: ClientHello messages from the TLS handshake, and the number of successfully established HTTPS sessions.

Applications on both platforms only specify TLS version 1.0 as the highest supported version in their ClientHello messages, while the built-in browsers support version 1.2. For this reason only ClientHello messages specifying a supported TLS version of 1.2 are collected.

The platform itself, built in components or services, also sends out data. This data is in almost every case similar to the traffic from applications. The one exception found in section 5.3 is the CDN from Akamai on iOS which supported TLS version 1.2 traffic. Traffic destined to Akamai is therefore filtered from the collected data.

The result is that only ClientHello messages from browsers are collected. For established HTTPS streams, the requests are inspected to reveal the User-Agent value in the headers as this can be used to determine of the requests comes from a browser or an application.

The resulting data shows that a device where a faulty certificate has been accepted for one connection, all following attempts to establish HTTPS connections will succeed. Established HTTPS connections included both search queries and credentials such as passwords. The data sample is too small to see if users are less inclined to accept a faulty certificate during a login.

![Figure 12: Devices with attempts and successes in establishing an HTTPS connection.](image)

In total attempts to establish an HTTPS connection was observed from seven different devices. Five of those did accept the faulty certificate and continued with the connection. These numbers are shown in figure 12.

18https://github.com/ephracis/device_sniffer
Using an adjusted Wald interval it can be determined that, with 95% confidence, between 24.98% and 84.25% of users will accept a faulty certificate.

7 Conclusions

This paper has looked at the state of communication security on smartphones. The platforms have been evaluated by the security tools they provide and how they educate developers both about those tools and about security in general.

The resulting work from the developers was then evaluated. Applications on Android were scanned by the hermes script using static code analysis and a smaller selection of the most popular free applications were manually inspected using traffic analysis.

Lastly, user behavior was evaluated using a rogue hotspot where connected users’ traffic was intercepted with a faulty certificate.

7.1 Platforms

The methods for trusting a specific certificate are similar on both platforms. On Android it is done by creating a custom TrustManager and using it during the connection, while on iOS the certificate is added to the KeyChain which is part of the platform. In both cases the certificate is trusted by the application and only that application. Other applications, or the platform itself, will not trust the self-signed certificate.

On Android the code for trusting a certificate is available on the documentation website, while the code for iOS is found as part of a larger code sample.

The documentation on Android is smaller in size and more to the point than the documentation on iOS. While the iOS documentation is less structured, it contains a lot more information and spans all the way from specific API references to general guides on security best practices. The iOS documentation website presents all documents as a long list where many types of documents are mixed.

The iOS documentation makes up for this by using a lot of references between documents. Developers are, for example, constantly reminded of the importance of security when implementing network communication in their applications.

The Android documentation, on the other hand, does not contain many links between the network section and the security section. This means that developers looking to implement network communication in their applications are not suggested to consider how to make the communication secure. Instead the developer must explicitly seek out the security section by navigating to it from the sidebar.

Warnings are more common in the documentation for Android than the documentation for iOS. These warnings are however of a general nature, they warn about common mistakes, pitfalls, and bad advice from third party websites. There are no warnings about specific vulnerabilities which has been discovered in the Android APIs.

Code samples for iOS are easy to find as they are available on the same location as the documentation. They can be downloaded directly via the website and there are a lot of different types of samples. By using the search filter, the code samples can quickly be found by category.

Code samples for Android are instead downloaded using an application which needs to be installed. This introduces some additional steps for retrieving code samples and could lower the chance of a developer downloading them.

There are much more code samples available for iOS than for Android and while the code samples for iOS mostly only use HTTP in their examples, they do mention HTTPS and either support it directly, or contain instructions on how to add such support. Some even advice the developer to use secure communication before using the code in production.

In short, the work done by both Google and Apple can be considered good enough when it comes to providing developers with enough tools for securing network traffic and educating their developers on how to use these tools as well as best practices.

More can however be done. Exposing developers more to security by cross linking between sections in the documentation can increase the chances of readers considering security early in the development process.

Making code samples more security focused may also increase the security of applications, since developers who copy code verbatim from those samples will end up with secure code. Having all code samples use secure network communication will also send the signal that security is very basic and a natural part of any network connected application.

Lastly, the platforms can make it a lot easier to trust self-signed certificates. Doing so will likely increase the chances that developers not using certificates signed by a trusted CA will properly add trust to that certificate only, instead of completely bypassing certificate validation.
7.2 Applications

Application developers on smartphones are using HTTPS more than they are using HTTP, which is a good sign. Unfortunately, sensitive information such as passwords and session data is exposed in around one in twenty applications. This is both due to failure to use proper protection such as HTTPS and due to failure to properly verify server certificates.

It is hard to spot a significant difference between Android and iOS. There is an 88% chance that applications on Android are more likely to use HTTPS. Looking at exposure of sensitive data, however, there is no difference between the two platforms. Applications are equally likely to expose users’ passwords by either failing to use HTTPS, or by using it without proper verification of the certificates.

Looking at both platforms, the probability that a given application will expose sensitive data lies between 4.31% and 10.12% with a 95% confidence.

All applications using HTTPS, except three web browsers and some traffic to the Akamai CDN, are using the old 1.0 version of TLS. This version contains several well documented attacks which has been proven practical against web browsers (there were several web browsers on Android observed of only supporting TLS version 1.0).

A large majority, 90.41%, of the applications on Android using HTTPS are using the RC4 cipher. This leaves them vulnerable to the RC4 bias attack which exploits a slight bias in some of the initial values of the keystream.

All other applications on Android, with the exception of the web browsers that supported TLS 1.0, and all applications on iOS, used AES in CBC encryption mode. This leaves them potentially vulnerable to the BEAST attack.

The BEAST attack requires a very specific attack model where the attacker need to inject custom data into the encrypted traffic stream. While possible on desktop web browsers, this might make the attack impractical against smartphone applications. The RC4 attack, on the other hand, works under a less restrictive attack model but instead needs a large number of repeated traffic in order to recover the plaintext.

None of these attacks were tested during this research. It is still uncertain how many applications are vulnerable to attack in practice. It should however be noted that while the attacks may not be practical, the vulnerabilities that the attacks exploit are still present in the traffic of the applications.

As such, every single application tested during the manual traffic analysis, on both Android and iOS, shows vulnerabilities that known attacks against web browsers exploit for either partial or full plain text recovery.

Correlations were found between the meta data of an Android application and its probability of containing code to bypass certificate validation. Applications with a higher rating are less likely to contain such code. So are applications with at least 100 downloads. The most secure applications are those with 10,000 to one million downloads.

Users wanting to decrease the risk of using applications without proper HTTPS certificate verification should thus use applications with a high rating and at least 100 downloads.

There are several improvement that can be done to increase the level security. First and foremost is to remove the ability to completely skip certificate verification. If this is not possible it should at least be made harder.

A high share of applications where certificates are not verified expose users’ passwords. Developers are sending sensitive data over unverified HTTPS a lot more than they are over plain HTTP (though the latter also occurs). This could be a result of a misconception about the role of authentication in HTTPS. It is thus important to educate developers of why encryption without authentication is highly insecure.

Another improvement that can be made is to add indicators of traffic type, similar to that in most web browsers. This would let users know if their data is sent over plaintext or over an encrypted channel. Such an indicator could be shown both when the application is running and on the application store before installing the application.

Platform providers could also use automatic tools similar to the hermes script to identify application containing insecure code. This automated testing could also be extended to test certificate verification and remove those applications that fail to properly verify a certificate from the application store.

More work should also be done to move HTTPS traffic away from the old TLS version 1.0 onto newer versions in order to prevent any future attacks that exploit the known vulnerabilities in that protocol.

7.3 Users

Lastly, this paper looked at how users behave when they are presented with a warning about a certificate which cannot be verified.
Sunshine et al. [32] found in a study in 2009 that 80% of users accept such faulty certificates. A similar experiment, described in section 6, was done at a train station where users were not aware of the experiment and using their own personal devices.

The hypothesis is that this might change users’ behavior. The result from the experiment contains a fairly small sample size but there is a 95% chance that at least one in four users will accept certificates.

The complete interval of users accepting faulty certificates, using a 95% confidence level, is between 24.98% and 84.25%. It should be noted that, due to how the rogue hotspot was configured, this experiment is biased toward students which might behave differently than the average population.

A very positive finding was that among applications which properly verified certificates, none gave the user the ability to bypass the verification and continue with the connection, as is possible in web browsers.

7.4 Future work

This paper has looked at communication security in a wide perspective and how it is used on modern smartphones. The data sample from manual analysis is fairly small and an improvement to the confidence of the conclusions drawn in this paper can be done by testing more applications.

The hermes script used for static code analysis can also be improved in several ways. It could be extended to support a similar scan on applications for iOS and other platforms. While reverse engineering is more complex on iOS than on Android, due to the lack of tools such as Androguard, it is still possible.

The hermes script could also be improved to identify the type of traffic each application uses. It could even possibly download and run the applications inside an emulator in order to profile the traffic. Some manual work would still be needed however in order to navigate the interface while the application is running.

The social experiment in section 6 can also be done on a larger scale in order to collect more data and thus get a higher statistical confidence. If indicators such as those previously mentioned would be presented to some users, their effectiveness could be measured.

Lastly, this paper only looked at applications on Android and iOS. While those two are the most popular platforms, several other exists. This research could be extended to include Windows Phone, Firefox OS, BlackBerry, Tizen, and Ubuntu for phones.

8 References


[10] Italo Dacosta, Mustaque Ahamad and Patrick Traynor. ‘Trust No One Else: Detecting MITM Attacks against SSL/TLS without Third-Parties’. In: Computer Secur-


Angelo Prado, Neal Harris and Yoel Gluck. ‘SSL, Gone In 30 Seconds’. In: Blackhat Conference 2013. Las Vegas, USA: Blackhat, 2013.


Figure 13: Usage of the Internet permission.
Figure 14: Distribution of verifier types over categories.
## Appendix B  Tables

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
<th>Internet</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>News and Magazines</td>
<td>1169</td>
<td>1158</td>
<td>99.06%</td>
</tr>
<tr>
<td>Social</td>
<td>1360</td>
<td>1307</td>
<td>96.10%</td>
</tr>
<tr>
<td>Game</td>
<td>2717</td>
<td>2610</td>
<td>96.06%</td>
</tr>
<tr>
<td>Shopping</td>
<td>841</td>
<td>794</td>
<td>94.41%</td>
</tr>
<tr>
<td>Comics</td>
<td>907</td>
<td>848</td>
<td>93.50%</td>
</tr>
<tr>
<td>Media and Video</td>
<td>2008</td>
<td>1871</td>
<td>93.18%</td>
</tr>
<tr>
<td>Travel and Local</td>
<td>1579</td>
<td>1471</td>
<td>93.16%</td>
</tr>
<tr>
<td>Sports</td>
<td>1613</td>
<td>1488</td>
<td>92.25%</td>
</tr>
<tr>
<td>Business</td>
<td>1367</td>
<td>1246</td>
<td>91.15%</td>
</tr>
<tr>
<td>Transportation</td>
<td>889</td>
<td>799</td>
<td>89.88%</td>
</tr>
<tr>
<td>Weather</td>
<td>867</td>
<td>773</td>
<td>89.16%</td>
</tr>
<tr>
<td>Lifestyle</td>
<td>2336</td>
<td>2078</td>
<td>88.96%</td>
</tr>
<tr>
<td>Books and Reference</td>
<td>2788</td>
<td>2467</td>
<td>88.49%</td>
</tr>
<tr>
<td>Finance</td>
<td>976</td>
<td>862</td>
<td>88.32%</td>
</tr>
<tr>
<td>Widgets</td>
<td>2299</td>
<td>2015</td>
<td>87.65%</td>
</tr>
<tr>
<td>Entertainment</td>
<td>2705</td>
<td>2352</td>
<td>86.95%</td>
</tr>
<tr>
<td>Education</td>
<td>2890</td>
<td>2500</td>
<td>86.51%</td>
</tr>
<tr>
<td>Medical</td>
<td>1125</td>
<td>968</td>
<td>86.04%</td>
</tr>
<tr>
<td>Photography</td>
<td>1583</td>
<td>1359</td>
<td>85.85%</td>
</tr>
<tr>
<td>Communication</td>
<td>2081</td>
<td>1785</td>
<td>85.78%</td>
</tr>
<tr>
<td>Health and Fitness</td>
<td>2061</td>
<td>1768</td>
<td>85.78%</td>
</tr>
<tr>
<td>Music and Audio</td>
<td>2428</td>
<td>2004</td>
<td>82.54%</td>
</tr>
<tr>
<td>Productivity</td>
<td>2142</td>
<td>1618</td>
<td>75.54%</td>
</tr>
<tr>
<td>Tools</td>
<td>2592</td>
<td>1931</td>
<td>74.50%</td>
</tr>
<tr>
<td>Libraries and Demo</td>
<td>823</td>
<td>555</td>
<td>67.44%</td>
</tr>
<tr>
<td>Live Wallpaper</td>
<td>2604</td>
<td>1598</td>
<td>61.37%</td>
</tr>
<tr>
<td>Personalization</td>
<td>2604</td>
<td>1452</td>
<td>55.76%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>38229</strong></td>
<td><strong>33101</strong></td>
<td><strong>86.59%</strong></td>
</tr>
</tbody>
</table>

Table 1: Distribution of apps with Internet permission over categories.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Apps with insecure code</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>1331</td>
<td>46.78%</td>
</tr>
<tr>
<td>1-2</td>
<td>60</td>
<td>38.46%</td>
</tr>
<tr>
<td>2-3</td>
<td>415</td>
<td>36.12%</td>
</tr>
<tr>
<td>3-4</td>
<td>1636</td>
<td>33.29%</td>
</tr>
<tr>
<td>4-5</td>
<td>4638</td>
<td>31.22%</td>
</tr>
</tbody>
</table>

Table 2: Apps with insecure code grouped by their rating, and share of such applications in that group.

<table>
<thead>
<tr>
<th>Downloads</th>
<th>Apps with insecure code</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-100</td>
<td>1336</td>
<td>45.17%</td>
</tr>
<tr>
<td>100-1,000</td>
<td>2747</td>
<td>32.35%</td>
</tr>
<tr>
<td>1,000,000-10,000,000</td>
<td>2986</td>
<td>30.67%</td>
</tr>
<tr>
<td>1,000,000,000-10,000,000,000</td>
<td>998</td>
<td>36.76%</td>
</tr>
<tr>
<td>10,000,000,000+</td>
<td>13</td>
<td>40.62%</td>
</tr>
</tbody>
</table>

Table 3: Apps with insecure code grouped by their download number, and share of such applications in that group.
Table 4: Apps with insecure code grouped by the year they were released or last updated, and share of such applications in that group.

<table>
<thead>
<tr>
<th>Year</th>
<th>Apps with insecure code</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>1</td>
<td>1.56%</td>
</tr>
<tr>
<td>2011</td>
<td>46</td>
<td>11.76%</td>
</tr>
<tr>
<td>2012</td>
<td>279</td>
<td>19.57%</td>
</tr>
<tr>
<td>2013</td>
<td>7686</td>
<td>35.17%</td>
</tr>
<tr>
<td>Unknown</td>
<td>68</td>
<td>35.23%</td>
</tr>
</tbody>
</table>