



Sensitive Data in Smartphone Applications: Where Does It Go? Can It Be Intercepted?

Eirini Anthi^(✉) and George Theodorakopoulos

School of Computer Science and Informatics, Cardiff University, Cardiff, UK
{anthies,theodorakopoulos}@cardiff.ac.uk

Abstract. We explore the ecosystem of smartphone applications with respect to their privacy practices towards sensitive user data. In particular, we examine 96 free mobile applications across 10 categories, in both the *Apple App Store* and *Google Play Store*, to investigate how securely they transmit and handle user data. For each application, we perform wireless packet sniffing and a series of man-in-the-middle (MITM) attacks to capture personal identifying information, such as usernames, passwords, etc. During the wireless packet sniffing, we monitor the traffic from the device when a specific application is in use to examine if any sensitive data is transmitted unencrypted. At the same time, we reveal and assess the list of ciphers that each application uses to establish a secure connection. During the MITM attacks, we use a variety of methods to try to decrypt the transmitted information.

The results show that although all tested applications establish a secure TLS connection with the server, 85% of them support weak ciphers. Additionally, 60% of iOS and 25% of Android applications transmit unencrypted user data over the Wi-Fi network. By performing a MITM attack we capture the username, password, and email in various apps, e.g. Instagram, Blackboard, Ebay, and Spotify. We manage to bypass certificate pinning in 75% of the iOS applications, including Facebook. Finally, we observe that data is being forwarded to third party domains (mostly to domains that belong to Google and Apple).

Keywords: Mobile security · Man-in-the-middle attacks
Wireless network security · Network sniffing · SSL/TLS

1 Introduction

In the last decade, the number of smartphone users has increased dramatically [36]. Smartphones are Internet-enabled devices with an operating system (e.g. iOS, Android, Windows), capable of executing a variety of applications. Most of these devices are also equipped with voice control functionality, a camera, a Wi-Fi antenna, Bluetooth, and GPS. Due to their capabilities, smartphone owners not only use their devices to communicate but also to perform important

everyday life activities. Such activities include researching a health condition, accessing education resources, navigating, and managing their money [34].

Most of the time users are required to share personal information with the mobile applications they use. However, it is often not clear to them how exactly these applications handle their personal data. A study by Boyels et al. [9] showed that 54% of smartphone users decided not to install an application once they discovered how much personal information they would need to share. Additionally, 30% of the users uninstalled an application that was already on their mobile phone when they learned it was collecting personal information they did not wish to share. The same study also showed that users are particularly sensitive about location data, with 19% of the users turning off the location tracking feature on their phone due to concerns about who could possibly access this information.

The rapid growth of the number of smartphone users has led to the increase of security threats related to smartphones. According to ENISA (European Union Agency for Network and Information Security), the number one threat is the leakage of data [13], which can happen in various ways: When a smartphone gets lost or stolen, its memory or removable media are unprotected, allowing an attacker to access the user's data [13]. Moreover, most of the applications used on a smartphone device will require the user to change their privacy settings in order to allow the application to access sensitive information such as contacts, photographs, etc. Many of these applications have been reported for sharing users' personal information with third parties without their consent. A recent study by Zang et al. [20] showed that 73% of Android and 47% of iOS applications shared personal information with third parties, including email addresses and location data. Finally, there is data loss that can occur when a smartphone is connected to Wi-Fi [22].

Although many smartphone users are aware that the mobile applications they use may share their personal data with third parties, many do not realise how often this happens [10]. Specifically, a recent survey [35], showed that many users are completely unaware of the risks that come when they share their personal data over a Wi-Fi connection, and particularly over public Wi-Fi networks. The most severe threat is the unauthorized access to their device which can lead to identity theft and compromised bank accounts [35].

This paper examines in depth the data leakage that occurs when users share personal information with various mobile applications over a Wi-Fi connection. Such information includes usernames, passwords, search terms, and location/geo-coordinates data. Additionally, we examine how these applications handle a user's personal information by observing the type of data they might share with third parties. Finally, we investigate methods to avoid data leakage. We perform tests on both *Android* and *iOS* devices; as they have different operating systems, we expect their behavior as to how they transmit and handle user data to differ.

The rest of the paper is organized as follows: Sect. 2 presents related work. Section 3 describes the experimental set up. Sections 4, 5, and 6 describe the main experiments and their results. Section 7 discusses the findings and evaluates the research. Finally, Sect. 8 covers the conclusion and future work.

2 Related Work

Previous studies have mainly focused on investigating the types of sensitive data that various mobile applications share with third parties, using dynamic analysis to capture mobile network traffic [6]. The major disadvantage of this approach is that requires human intervention, which can limit the scaling of the experiment. Various methodologies fall under this approach and have been used successfully in the past.

For instance, Rao et al. [32] used a Virtual Private Network (VPN) to monitor mobile traffic, involving tools such as *Meddle*. They showed that a significant number of Apple *iOS* and Google *Android* applications shared sensitive information such as emails, locations, names, and passwords as plaintext. A different way to observe network traffic is directly on the device. The *TaintDroid* application [4] for the *Android* platform allows users to track how private information is obtained and released by mobile applications in real time. A study by Enck et al. showed that 15 applications sent user location data to third parties and 30 sent the unique phone identifier, phone number, and SIM card serial number. Zang et al. [20] used a third method to monitor network traffic, during which they performed a man-in-the-middle attack over the Wi-Fi network that the device was connected. They showed that a very large percentage of mobile applications shared personal data with third parties and connected to unknown domains.

Another study which used the same method as [20] was that of Thurm and Kane [38]. This study revealed that a music *iOS* application shared personal information with eight different domains. Furthermore, the Federal Trade Commission [16] applied the same method to research the behavior of 15 fitness applications. The results of this study showed that 12 of the applications transmitted identifying information to 76 third party domains.

These studies focus on investigating the types of sensitive data that various mobile applications share with third parties. However, how securely these applications transmit this data over Wi-Fi networks has not yet been examined.

In this paper, we build on previous work by testing 96 free applications that require personal information. We investigate how user sensitive data is transmitted and handled, using wireless packet sniffing and dynamic analysis with man-in-the-middle attacks over a Wi-Fi network.

3 Experimental Setup

3.1 Selecting Mobile Applications

The Google *Play Store* for *Android* and the Apple *App Store* for *iOS* are the two largest distribution channels for mobile applications [41], which is why we focus on these two platforms. From a total of 96 applications that we test, 51 are *iOS* and 45 are *Android*. These are the most popular applications as of January/February 2016 that handle sensitive user data, across 10 different categories: Business, Finance, Food and Drink, Games, Health and Fitness, Music,

Productivity, Shopping, Social Networking and Travel. We test the *iOS* applications on an *iPhone 6/iOS v9.0.1* and the *Android* applications on a *Motorola Moto e/ Kit Kat v4.1*. Table 1 in the Supplemental Material¹ contains all the applications that we examine in this research.

3.2 Testing the Mobile Applications

In order to test each application we manually simulate a typical use for 10 to 15 min. The time spent on each application varies and exclusively depends on its type. During the simulation we explore the basic functions of the application. These include: create a user account, search using various keywords, perform actions that require personal identifying data, and complete a level of a game. We record specific keywords and personal user data that are used during each simulation. We then search for these keywords and personal data in the captured communications. To ensure the integrity of the captured data and to avoid possible interference from other applications, we take the following measures: during testing only the tested application is open and no other. We achieve this by terminating all other applications and by observing whether any data is transmitted, while no applications are open. For each application, we allow all requested permissions, such as for sharing location data, except for push notifications. The reason we disable push notifications is because they keep sending data in the background even after the application is closed [15]. This would result in capturing data not only from the application being tested at any single time, but also from any previously tested applications that enabled push notifications.

4 Experiment 1: Examining Network Data Following SSL Employment

To identify if any of the applications transmit unencrypted data over the Wi-Fi network, we perform wireless packet sniffing using *Wireshark* [26]. During this process we passively monitor the mobile traffic from the smartphone. After configuring *Wireshark* to monitor mobile traffic from the smartphone, we start using an application. For each application, we test all the captured packets for user sensitive data using *Wireshark*'s built-in filter functionality.

If the mobile applications do not employ the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) protocol [33], the data that gets transmitted is not encrypted, hence it can simply be intercepted by performing passive network sniffing on the operating channel. If the SSL/TLS is employed, the transmitted data is encrypted and no third party is able to eavesdrop on or interfere with any of the transmitted messages [29]. As a result, for any application that employs SSL, we are unable to read or modify any of the transmitted messages. However,

¹ The supplemental material has been placed in our institutional repository due to space constraints. It can be accessed at this link: <http://orca.cf.ac.uk/id/eprint/101448>.

the SSL connection can be weakened in a number of ways and hence it is possible to decrypt the transmitted data.

In order for an SSL connection to be established, the client and the server make use of cipher suites. A cipher suite consists of a key exchange algorithm, a signature algorithm, a block cipher algorithm, and a hashing algorithm which computes the authentication key [29] (see Fig. 1). There is a variety of cipher suites available that provide different levels of security. The choice of cipher suites is crucial as they can compromise the security of the communication. Even if one of the listed cipher suites is cryptographically insecure, it is enough to break the secure connection between the client and the server and hence intercept the communication. This is possible via the *TLS Protocol Downgrade* attack [25] and it is one of the ways in which the SSL/TLS connection can be weakened.

```
[SSL/TLS]_[key exchange]_[signature algorithm]_WITH_[block cipher]_[authentication hash]
```

Fig. 1. Format of a cipher suit

Via *Wireshark* we are able to view the list of the cipher suites that each application supports to establish a secure connection with the server and as a result we can assess how secure they are. To achieve this we use data from the *O-Saft* [28] tool, which is used to inspect information about SSL/TLS certificates and tests the SSL/TLS connection, according to a given list of cipher suites. The code within *O-Saft* contains an evaluation of the strength of different cipher suites. To rate a cipher suite as weak or strong, the script examines the level of security of the individual algorithms (including the length of the key they use - if applicable) that compose the cipher suit. The script contains all possible combinations of cipher suites followed by a description of the level of their security, described as weak, medium, and high. Immediately afterwards, it displays a break down of each cipher, which explains the algorithms they contain and their key lengths in further detail.

Results: All the tested mobile applications for both *iOS* and *Android* platforms employ the latest version of SSL to establish a secure channel for communication. As a result, although we are able to capture the transmitted data, it is not possible for us to read it because it is encrypted. The only case in which we have the opportunity to capture transmitted data in plaintext is when we test the mobile browsers, *Safari* on the *iPhone* and *Google Chrome* on the *Motorola*, and perform requests that do not require a secure connection.

We examine and assess the cipher suites in 51 *iOS* applications, and we find that 45 use the same set of 26 cipher suites. From these 26 suites, 4 are considered to be weak and should not be used. Only 6 of the tested applications use less than 26 suites and do not support any weak suites (see Fig. 2). From the 45 *Android* applications, 27 use the same set of 35 cipher suites, of which 4 are considered insecure. Moreover, 11 of the applications use less than 35 cipher suites and from

these only 6 do not support any insecure suites. Just 3 applications use more than 35 suites and only 1 does not support weak cipher suites. Finally, it was not possible to capture the *ClientHello* message for 4 applications and as a result their cipher suites could not be assessed (see Fig. 3).

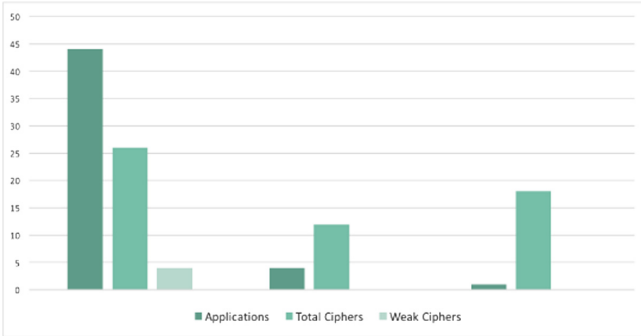


Fig. 2. Number of cipher suites that *iOS* applications support and how many of these are considered to be weak.

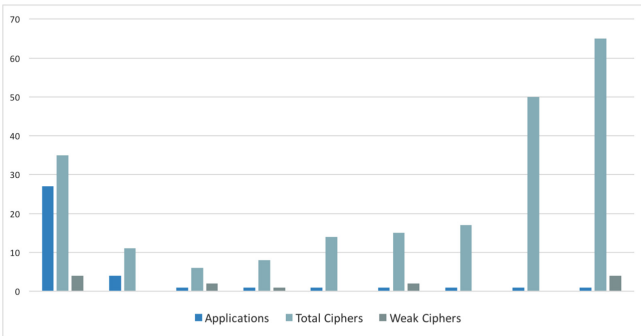


Fig. 3. Number of cipher suites that *Android* applications support and how many of these are considered to be weak.

Table 3 in the Supplemental Material shows in detail the number of cipher suites each application uses and how many of these are considered to be weak. For both systems we find that the applications support the same 4 insecure cipher suites, which are:

1. TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
2. TLS_ECDHE_RSA_WITH_RC4_128_SHA
3. TLS_RSA_WITH_RC4_128_SHA
4. TLS_RSA_WITH_RC4_128_MD5

The order in which the suites appear in the *ClientHello* message denotes the client's preferred suites (with the client's highest preference first). In the *ClientHello* message, for all *iOS* applications, we observe that these 4 suites are at the bottom of the list, as opposed to the *Android* applications where the suites are found to be at the top of the list, which shows that these are the client's most preferred suites. Therefore, in the first case, the four weak cipher suites are the least preferred suites by the client and are unlikely to be used to establish a secure connection [1]. In the second case, the weak suites seem to be the client's most preferred suites. If the server accepts the client's preferences (the server is free to ignore the client's order and can pick the cipher suite that thinks it is best [1]) a connection will be established using one of these insecure suites, making the application vulnerable to MITM attacks. Regardless of the order in which these weak cipher suites appear in the application's *ClientHello* messages, they should not be used, as a *TLS Downgrade Attack* [25] could be used against them.

5 Experiment 2: Examining Network Data After Bypassing SSL

To examine how various applications transmit and handle user data other than sniffing the packets on the wireless network, we also use dynamic analysis with MITM attacks. The MITM attack is a technique used to intercept the communication between two systems, in this case between the client (application) and the server [27].

There are many tools that can be used to perform such an attack. Specifically, in this paper we use *Burp Suite* [37] and *mitmproxy* [8]. These also help us identify only HTTP-based traffic. We note that a recent study by Raoa et al. [32] showed that TCP flows (HTTP/HTTPS) are responsible for over 90% of the total traffic volume. Finally, in order to perform the attacks described above, we need to setup a Wi-Fi hot-spot on a computer that runs these tools and connect the smartphone device to the Internet via this hot-spot (Fig. 4).

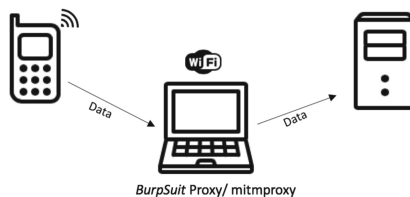


Fig. 4. Man-in-the-middle attack using *Burp Suite* and *mitmproxy*.

5.1 Man-in-the-Middle Attack Using *Burp Suite*

To examine if an application is accepting self-signed certificates, it is necessary to configure the smartphone to use a proxy. In this case we use *Burp Suite*, which generates a self-signed certificate and presents it to the client. We then monitor the behavior of the application in use and observe if it functions as expected. Additionally, we check if we are able to capture any HTTPS traffic on the proxy software. The steps of the procedure are described below [39]:

1. We ensure that the smartphone does not have any existing custom proxy certificates in its trust store.
2. On the computer, we disable the firewall and start the *Burp Suite* proxy. It is necessary to configure it to listen to all external network interfaces by specifying the port and address.
3. Then we configure the smartphone device to use the proxy. (Settings, Wi-Fi, we choose the desired Wi-Fi network, select HTTP Proxy Manual). The IP address and port of the proxy are the same to the computer in use.
4. Finally, we launch the application we want to test and simulate a typical use, while we monitor the proxy to detect if any HTTPS data is being intercepted.

If *Burp Suite* is able to intercept HTTPS traffic from the device without us having to install the proxy's certificate on the device's trust store, we know that the application does indeed accept self-signed certificates and is vulnerable to eavesdropping and modification via MITM attacks [39].

Results: We find that none of the applications for both platforms accept the unverified certificate that *Burp Suite* generates, and they prompt us with a message as shown in Fig. 5. As a result, we are not able to capture any of the HTTPS traffic that occurs during the simulation of a typical use for each application.

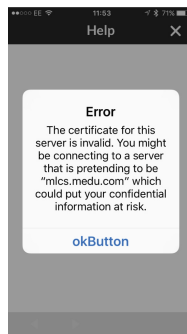


Fig. 5. *Blackboard* application rejecting *Burp Suite*'s self-signed certificate

5.2 Man-in-the-Middle Attack Using *mitmproxy*

On applications that do not accept self-signed certificates, we are not able to capture the encrypted traffic that occurs from the device using the previous method. In order to overcome this, we perform a MITM attack using *mitmproxy*.

Once again, we configure the smartphone to use the proxy. However, this time we install the proxy's certificate in the device's trust store. *mitmproxy* contains a Certificate Authority (CA) implementation and is able to generate digital certificates [24]. Furthermore, to make the client (device) trust certificates issued by *mitmproxy*, we register it manually on the device as a trusted CA. It is necessary to emphasize that this method will only work if the application does not employ certificate pinning [12]. More details about this mechanism and how to bypass it are in Sect. 6.

To intercept traffic with the *mitmproxy* we follow the steps below [23]:

1. We start *mitmproxy* and configure the device to use it by setting the correct proxy details (port and IP address).
2. We then open the browser on the smartphone and visit www.mitm.it.
3. We select the relevant icon and follow the instructions, as to how to install the proxy's certificate in the device's trust store. When the installation is completed, we open an application and start observing the *mitmproxy*'s screen for HTTPS traffic.

In the *mitmproxy*'s main screen, we are able to view the mobile traffic that occurs when an application is in use. *mitmproxy* displays the full flow summary, including the methods used and the full *Uniform Resource Identifiers (URIs)* of the HTTP/HTTPS requests. By selecting one of the requests, the software allows us to inspect and manipulate the data it contains [24]. If the application is not using any encryption on the transmitted data, we are able to view it as plaintext. Therefore, this method helps us identify if the applications transmit unencrypted information over the network and examine if they send any of it to unknown third parties. To analyze further the captured communications, we export all captured data to a text file and use a *Python* script to search in it for any user sensitive data that might have been transmitted in plaintext. Specifically, the data we look for includes: Personal Identifying Information (PII) such as names and passwords, search terms, and geo-coordinate data, including longitude and latitude values. In Table 1, we present all the types of user data that the script looks for in the text files. The complete list of the keywords that are used throughout the simulations and therefore we look to find in the captured data, can be found in Table 2 in the Supplemental Material. Moreover, in our *Python* script we identify all the URIs of the requests that the application performed POST requests for. This way we are able to discover if any of the applications transmit personal user data to unknown domains.

In order to ensure that our results are reliable, every time that the script identifies an occurrence of a keyword within a text file, we manually inspect the findings to confirm that they are correct and identify any further information. For instance, if the script finds a match for the string "1990", we manually examine the result to ensure that the finding is indeed the user's year of birth

Table 1. Types of user data.

Categories of data	Data types
Behavior	Employment (job searches)
	Medical
	Private messaging (chats, texts, etc.)
	Searching
Location	Latitude
	Longitude
PII	Address
	Age
	Date of birth
	Device information (e.g. Device ID)
	Email address
	Gender
	Name
	Password
	Post code
	Telephone number
	Username

and not a part of some other information such as long integer [20]. This process is also necessary in order to discover the destination domain, of the data that is transmitted and identified as plaintext.

Results: In order to perform this MITM attack it is necessary to install the certificate that *mitmproxy* generated in the device's trust store. After we complete this procedure, we observe that the *Android* device displays a warning message (see Fig. 6) to inform us that an unauthenticated certificate is currently being used. In contrast, on the *iOS* device we do not get any warnings about the fake certificate. Nevertheless, at this point we are able to capture HTTPS traffic from both devices, hence we start testing the applications, the results of which are presented in the following sections.

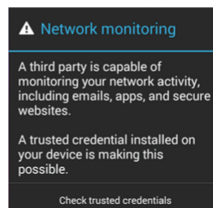


Fig. 6. Warning message on the *Android* device, regarding the *mitmproxy*'s fake certificate.

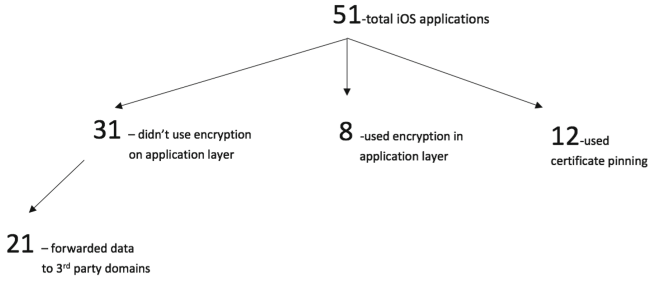


Fig. 7. The number of *iOS* applications that use encryption in the application layer, employ certificate pinning, and transmit sensitive data to 3rd party domains.

Results for *iOS* Applications: From the 51 applications, we find that 30 transmitted the data unencrypted over the network, of which 20 forward it to third party domains. Just 8 of the applications encrypt user data in the application layer (i.e. before passing it to SSL), therefore although we can capture the transmitted data, we are unable to read it. Finally, 12 applications employ certificate pinning and do not function at all (see Fig. 7), claiming that there is a problem with the network.

Table 5 in the Supplemental Material shows the sensitive data that we capture for each application and the domains that each one forwards data to. In the same table we mark applications that employ certificate pinning with an xmark and use “n/a” for data that is not being forwarded to any third party domains.

The Burger King, Indeed Jobs, Lose it!, and Ebay applications transmit the most unencrypted user data, which includes: usernames, passwords, emails, location, gender, and search terms. Additionally, we manage to capture usernames and passwords for Spotify, Blackboard, Instagram, and EasyJet. The applications that forward the most data to third party domains are Indeed Jobs and Burger King. Gaming applications mainly transmit and share information about the device such as: phone model, screen size, etc. Moreover, the third party domains that receive the most sensitive user data are [googleanalytics.com](https://www.google.com/analytics/), [googleservices.com](https://www.google.com/services/), and [apple.com](https://www.apple.com/). Figure 8 shows the types of data that the 20 *iOS* applications share with third parties.

Being able to capture the username, password, and email for Instagram, EasyJet, Blackboard, Ebay, and Spotify is a vulnerability. If an unauthorised person logs into these applications using these credentials, they could have access to much more sensitive information such as PayPal, bank accounts, home address, passport details, etc. Therefore, we decided to report our observations to each of the application’s development teams as per the *Responsible Disclosure*² procedure. Facebook (for Instagram), Spotify, and Blackboard replied to us thanking us for reporting this issue, confirming that it is indeed a security flaw.

² This procedure involves privately notifying affected software vendors of vulnerabilities. The vendors then typically address the vulnerability at some later date, and the researcher reveals full details publicly at or after this time [18].

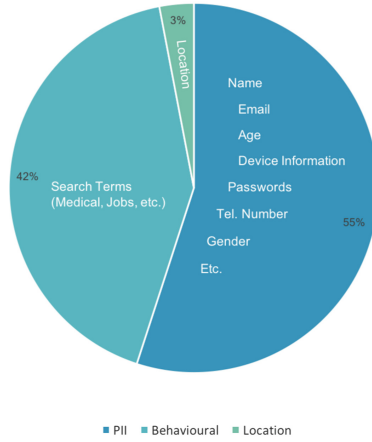


Fig. 8. The number of *iOS* applications that use encryption in the application layer, employ certificate pinning, and transmit sensitive data to 3rd party domains.

Results for *Android* Applications: From the 45 applications that we examine, 11 do not use any encryption in the application layer, hence the data gets transmitted unencrypted over the Wi-Fi network. Only 9 applications use encryption on the actual user data, so although we are able to capture the network traffic we are not able to read it. Furthermore, 25 applications employ certificate pinning and do not function during this process (see Fig. 9). Table 6 in the Supplemental Material shows the transmitted sensitive data that we capture for each *Android* application and also the third party domains to which it is being sent.

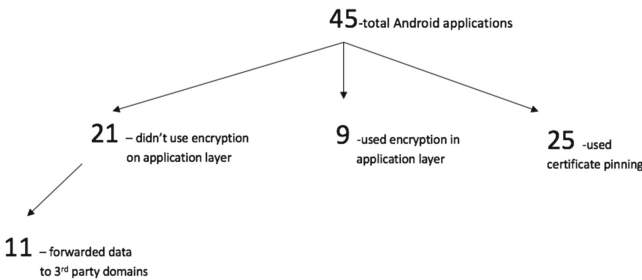


Fig. 9. The number of *Android* applications that use encryption in the application layer, employ certificate pinning, and transmit sensitive data to 3rd party domains.

Ebay, Gumtree, and [Booking.com](#), are the only applications that transmit unencrypted usernames and passwords. Domino’s Pizza, Gumtree, and [Booking.com](#) share with third parties all the terms that were searched for in the

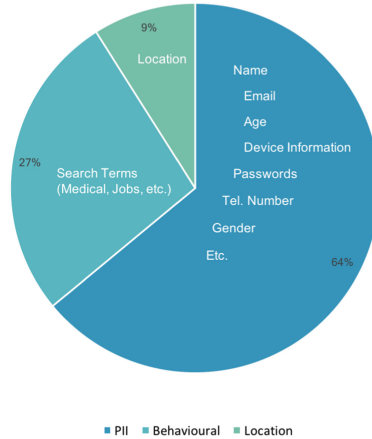


Fig. 10. The number of *Android* applications that use encryption in the application layer, employ certificate pinning, and transmit sensitive data to 3rd party domains.

application. Finally, location data is only shared by Just Eat and gaming applications mainly transmit and share device information. The third party domains that receive the most user sensitive data are googleads.com and apple.com. Figure 10, shows the types of data that the 11 *Android* applications share with third parties.

6 Experiment 3: Bypassing Certificate Pinning

Certificate pinning is a technique used widely in mobile applications to prevent the possibility of the device’s trust store being compromised, by manually installing unverified certificates [12]. Specifically, this technique pins the certificate that the server uses in the application’s source code, forcing it to ignore the device’s trust store. As a result, it will only establish a connection to hosts signed with certificates that are pinned in the application’s source code. To applications that employ this mechanism, we use *iOS SSL Kill Switch* to attempt to bypass it.

We perform this procedure only on *iOS* applications, and we are required to *Jailbreak/Rooting* [11] the tested device. This allows us to remove all the software restrictions of *Apple*’s operating system and grants us access to the *iOS* file system and manager. As a result, we are able to download extra items that are unavailable on the official *Apple App Store* [11].

After *jailbreaking* the *iPhone 6* following the instructions on [30], we gain access to *Cydia*, the unofficial *iOS App Store*. From there we can download and install in the device *iOS SSL Kill Switch* [2]. This tool disables the certificate validation process on the client side (the device), leaving it exposed to MITM attacks. Having installed and enabled *iOS SSL Kill Switch*, we use *mitmproxy* following the method described in the previous Sect. 5 to check if we can capture any HTTPS traffic.

Results: We find that this tool is effective on 75% of the applications, allowing us to capture the traffic that is transmitted while we are testing them. The remaining 25% of the applications are able to detect that the device is *Jailbroken* and do not operate (e.g. banking & social media applications).

7 Discussion

We perform wireless packet sniffing to investigate if any of the mobile applications transmit data unencrypted over the Wi-Fi network. Our results show that all the applications for both *iOS* and *Android* platforms use SSL to establish a secure channel for communication with the server. This protocol is fairly widely employed by developers, as it provides protection against passive eavesdropping [8]. Anyone performing wireless packet sniffing over the network will be able to capture the traffic, but they won't be able to read it as it is encrypted. SSL may provide privacy and data integrity between a client and a server, however it can be weakened and the cipher suites that applications use to establish this connection have an important role in this. We examine all the cipher suites that applications support in order to establish a secure connection and we find that the majority of them in both platforms (90% of the *iOS* and 80% of the *Android* applications) support four insecure cipher suites. These suites were the same for both operating systems:

1. TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
2. TLS_ECDHE_RSA_WITH_RC4_128_SHA
3. TLS_RSA_WITH_RC4_128_SHA
4. TLS_RSA_WITH_RC4_128_MD5

These cipher suites are considered to be weak mainly because they use the RC4 stream cipher. Even though RC4 is widely supported and preferred by most servers, it has been known to have a variety of cryptographic weaknesses, making it unable to provide a sufficient level of security [3, 19]. For this reason, according to the Internet Engineering Task Force (IETF), the RC4 algorithm is prohibited and clients must not include RC4 ciphers in their *ClientHello* message. Additionally, the MD5 hash algorithm is also known to have cryptographic weaknesses and cipher suites that employ it should not be used [14, 29]. A few of the reasons that applications support these suites although they are considered to be insecure and have been prohibited include: compatibility with most servers, simple design, and speed due to the reduced number of operations they need to perform [31]. Nevertheless, 85% of all the tested *iOS* and *Android* applications that support these suites, even though they use SSL, are considered to potentially be vulnerable to MITM attacks.

We also test the applications in order to investigate if they accept self-signed certificates. We find that none of the applications, for both *iOS* and *Android*, accept the self-signed certificate that *Burp Suite* proxy generates. This is an indication that accepting self-signed certificates is indeed a severe security issue

that developers are aware of, making the certificate validation processes as robust as possible [39].

Using *mitmproxy* we establish that approximately 60% of the *iOS* and 25% of the *Android* applications transmit and forward sensitive unencrypted data to third party domains. The most common data forwarded by applications to third party domains is Personal Identifying Information (PII) and Behavioral including: device information, email, name and search terms. For both platforms, gaming applications mainly transmitted and forwarded information about the device. A reason why PII and behavioural types of data are shared with third parties could be that this information is used by these organisations to develop targeted advertising [40]. The percentage of *Android* applications that share user data with third party domains seems to be significantly less than the percentage of the *iOS* applications. This is due to the fact that 20% of *Android* applications encrypt the actual user data and 56% employ certificate pinning. On the other hand, only 15% of the *iOS* applications encrypt the user data and only 23% employ certificate pinning. Therefore, for the applications that encrypt the data and use certificate pinning we are unable to investigate if they share sensitive information with third parties.

Comparing our results with a recent study by Zang et al. [20], which also investigated data sharing by applications, we can observe some differences. In the previous study, more applications shared location and other sensitive user data and very few employed certificate pinning. On the contrary, our results show that fewer applications share location and other sensitive user data with third parties. Additionally, the number of applications that use certificate pinning, specifically when it comes to *Android* applications, has increased dramatically. The overall increase in applications employing certificate pinning may be because, without it, data can be intercepted by installing fake certificates in the device's trust store [12]. Additionally, penetration testing recently performed on various mobile applications [20, 21] could also explain why more of them started using certificate pinning. The fact that significantly more *Android* applications employ certificate pinning compared to *iOS* is because certificate pinning is one of the many security enhancements introduced in the new firmware version, Android 4.2 [12].

The domains to which applications from both platforms send the most user sensitive data are: googleanalytics.com, googleservices.com, googleads.com, and apple.com. Previous studies [20, 32] have also found these domains to be dominant. This may be due to Google and Apple owning a variety of mobile advertisement networks and services such as AdMob, Google Analytics, Double CLick and iAds [5, 17].

Finally, we use *SSL Kill Switch* on a *Jailbroken* iPhone, in order to attempt to bypass certificate pinning on applications that employ it, and we successfully manage to do so in 75% of the applications. Finance applications (Barclays, PayPal, Pingit) detected that the device was *jailbroken* and did not operate. To conclude, *Jailbreaking* or *Rooting* the smartphone introduces security issues and unless the applications are designed to not operate in such a device, the user's data is in danger of being stolen.

Overall, the methods we choose to evaluate how securely mobile applications transmitted and handled user data over a Wi-Fi network are effective but have limitations. To begin with, all the methods we use require human intervention which limits significantly the number of applications that we are able to test. The MITM attacks we perform to both platforms, although they were able to provide us with valuable information about the applications certificate validation process and data sharing behaviour, require physical access to the device in order to install fake certificates. Therefore, even though we are able to intercept any transmitted sensitive data, these methods would be very difficult to apply in real life. Additionally, the tools we use to perform these attacks focus only on HTTP/HTTPS traffic, limiting the scope of the research. The *SSL Kill Switch* allows us to successfully bypass the certificate pinning mechanism; however, we need to *jailbreak* the iPhone. This is a very time consuming and insecure process. To analyse the captured data, we write a Python script to search for sensitive data in the captured communications text files. The script is very effective in analysing our data, however if these files were larger in size, Python would run very slowly and would not be the most appropriate language to use to implement it.

8 Conclusion and Future Work

Our study aims to explore and analyse how user data is transmitted and handled by various mobile applications. We select 51 *iOS* and 45 *Android* mobile applications and carry out 4 different experiments, while we simulate a typical use for each application. The results show that all applications use SSL protocol to establish a secure channel for communication with the server, which protects data from passive eavesdropping, specifically when transmitted over public networks. However, this does not mean that user data is secure, as our findings show that only a very small percentage of these applications encrypt the actual user data and approximately 85% of these applications support 4 weak cipher suites which make them vulnerable to MITM attacks. Moreover, our results show that 60% of the *iOS* and 15% of *Android* applications forward sensitive user data, mostly PII and Behavioral, to third party domains mainly owned by Google and Apple.

Although our research methodology has its limitations, we still manage to arrive at significant conclusions as to how securely user data gets transmitted and handled by various applications, over a Wi-Fi network. Additionally, two of the methods we use are designed to break or bypass the basic security mechanisms that developers employ, such as SSL and certificate pinning. This is proof that these security measures are not invulnerable. As a result, users need to become fully aware that their personal information can never be 100% secure and the only way to protect their privacy is to understand these security risks.

To expand on the results of this research, future study could focus on testing more applications from each category, for both operating systems. Non-TCP traffic could also be investigated for sensitive data leakage using *tcpdump*, which

monitors traffic that is not on TCP. To the applications that support weak cipher suites *TLS Downgrade Attack* could be performed, to explore if SSL can indeed be compromised this way. In this paper, we manage to apply tools to bypass certificate pinning only to *iOS* devices. Future studies could also *root* an *Android* device and then use *Android-SSL-TrustKiller* [7] to try to bypass certificate pinning in this operating system as well. Furthermore, tools that track the data-sharing behavior of applications directly from the smartphone device such as *TaintDroid* could be used to monitor both the operating system and the application. As a result, it would be possible to clearly distinguish any leakage that happens due to the application's activity and the background system processes [4, 20].

Additionally, paid applications could also be tested for data leakage. The results could then be compared to free applications in order to review any difference in the data sharing behavior. Finally, tools that limit data sharing, such as *Limit ad Tracking* and *Opt out of interest based ads*, can be employed to examine any differences in the activity of the applications.

References

1. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. <https://tools.ietf.org/html/rfc5246>. Accessed 05 Jan 2017
2. Diquet, A.: iOS SSL Kill Switch (2016). <https://github.com/iSECPartners/ios-ssl-kill-switch>. Accessed 20 Apr 2017
3. AlFardan, N.: On the Security of RC4 in TLS. <http://www.isg.rhul.ac.uk/tls/>. Accessed 25 Apr 2017
4. Appanalysis. Realtime Privacy Monitoring on Smartphones (2016). <http://www.appanalysis.org/index.html/>. Accessed 9 Apr 2017
5. Apple. Ad for Developers. Apple Developer. <https://developer.apple.com/iad/>. Accessed 03 May 2017
6. Ball, T.: The concept of dynamic analysis. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/SIGSOFT FSE -1999. LNCS, vol. 1687, pp. 216–234. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48166-4_14
7. Blanchou, M.: iSECPartners/Android-SSL-TrustKiller. Bypass SSL certificate pinning for most applications. <https://github.com/iSECPartners/Android-SSL-TrustKiller>. Accessed 03 May 2017
8. Boneh, D., Inguva, S., Baker, I.: SSL, MITM Proxy (2007). <http://crypto.stanford.edu/ssl-mitm>
9. Boyles, J.L., Smith, A., Madden, M.: Privacy and data management on mobile devices. Pew Internet Am. Life Project 4 (2012)
10. Carnegie Mellon University. Knowledge of Location Sharing by Apps Prompts Privacy Action (2015). <https://www.sciencedaily.com/releases/2015/03/150323132846.html>. Accessed 4 Apr 2017
11. Cohen, A.: The iPhone Jailbreak: A Win Against Copyright Creep. Time.com (2010)
12. Elenkov, N.: Certificate Pinning in Android 4.2 (2012)
13. ENISA. Top Ten Smartphone Risks (2016). <https://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1/top-ten-risks>. Accessed 4 Apr 2017

14. MIT Laboratory for Computer Science and RSA Data Security. RFC 1321 - The MD5 Message-Digest Algorithm. <https://tools.ietf.org/html/rfc1321>. Accessed 25 Apr 2017
15. Fox, M.A., King, P.F., Ramasubramani, S.: Method and apparatus for maintaining security in a push server. US Patent 6,421,781, 16 July 2002
16. FTC. Federal Trade Commission (2016). <https://www.ftc.gov/search/site/fitness~app>. Accessed 9 Apr 2017
17. Google. Monetize and Promote with Google Ads.Google Developers. <https://developers.google.com/ads/?hl=en>. Accessed 03 May 2017
18. Google. Rebooting Responsible Disclosure: A Focus on Protecting End Users. <https://security.googleblog.com/2010/07/rebooting-responsible-disclosure-focus.html>. Accessed 30 Apr 2017
19. Internet Engineering Task Force (IETF). RFC 7465 - Prohibiting RC4 Cipher Suites. <https://tools.ietf.org/html/rfc7465#section-1>. Accessed 25 Apr 2017
20. Zang, J., Dummit, K., Graves, J., Lisker, P., Sweeney, L.: Who knows what about me? A survey of behind the scenes personal data sharing to third parties by mobile apps (2015). <http://techscience.org/a/2015103001/>. Accessed 14 Feb 2017
21. Mense, A., Steger, S., Sulek, M., Jukic-Sunaric, D., Mészáros, A.: Analyzing privacy risks of mhealth applications. *Stud. Health Technol. Inform.* **221**, 41 (2016)
22. Cooney, M.: 10 Common Mobile Security Problems to Attack (2012). <http://www.pcworld.com/article/2010278/10-common-mobile-security-problems-to-attack.html>. Accessed 4 Apr 2017
23. mitmproxy. About certificates (2016). Accessed 20 Apr 2017
24. mitmproxy. How mitmproxy works (2016). Accessed 20 Apr 2017
25. Moeller, B., Langley, A.: RFC 7507: TLS fallback signaling cipher suite value (SCSV) for preventing protocol downgrade attacks (2015)
26. Orebaugh, A., Ramirez, G., Beale, J.: Wireshark & Ethereal Network Protocol Analyzer Toolkit. Syngress, Rockland (2006)
27. OWASP. Man-in-the-Middle Attack (2016). <https://www.owasp.org/index.php/Man-in-the-middle.attack/>. Accessed 18 Apr 2017
28. OWASP. O-Saft (2016). <https://www.owasp.org/index.php/O-Saft/>. Accessed 20 Apr 2017
29. OWASP. Transport Layer Protection Cheat Sheet (2016). https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet. Accessed 18 Apr 2017
30. Pangu. Pangu Jailbreak (2016). <http://en.pangu.io>. Accessed 20 Apr 2017
31. Paul, S., Preneel, B.: On the (in)security of stream ciphers based on arrays and modular addition. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 69–83. Springer, Heidelberg (2006). https://doi.org/10.1007/11935230_5
32. Raa, A., Kakhkib, A.M., Razaghpanahe, A., Tangc, A., Wangd, S., Sherryc, J., Gille, P., Krishnamurthyd, A., Legouta, A., Misloveb, A., et al.: Using the middle to meddle with mobile. Technical report, Northeastern University (2013)
33. Rescorla, E.: SSL and TLS: Designing and Building Secure Systems, vol. 1. Addison-Wesley Reading, Boston (2001)
34. Smith, A.: Us Smartphone Use in 2015. Pew Research Center, pp. 18–29 (2015). Accessed 1 Apr 2017
35. Statista. The Hidden Dangers of Public WiFi (2016). http://www.privatewifi.com/wp-content/uploads/2015/01/PWF_whitepaper_v6.pdf/. Accessed 5 Apr 2017
36. Statista. Number of Smartphone Users Worldwide from 2014 to 2019 (2016). <http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>. Accessed 1 Apr 2017

37. Stuttard, D.: Burp Suite (2007)
38. Thurm, S., Kane, Y.I.: Your apps are watching you. *Wall Str. J.* **17**, 1 (2010)
39. Chell, D., Erasmus, T., Colley, S., Whitehouse, O.: *The Mobile Application Hacker's Handbook*, 1st edn. Wiley, Hoboken (2015)
40. Varshney, U., Vetter, R.: Mobile commerce: framework, applications and networking support. *Mob. Netw. Appl.* **7**(3), 185–198 (2002)
41. Victor, H.: Android's Google play beats app store with over 1 million apps, now officially largest (2013). Accessed 16 Jan 2014