UNIVERSITY OF CAMBRIDGE

Tom Burrows

Detecting privacy leaks in Android messengers

Computer Science Tripos – Part II

Christ's College

May 13, 2022

Declaration

I, Tom Burrows of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed *Thomas Paul Murnane Burrows*

Date *13 May 2022*

Proforma

Candidate Number	2354C
Project Title	Detecting privacy leaks in Android messengers
Examination	Computer Science Tripos – Part II, June 2022
Word Count	3186[1]
Code line count	648[2]
Project Originator	Prof R. Anderson
Supervisor	Nicholas Boucher

Original Aims of the Project

To create a tool capable of logging and decoding data being transmitted by Android messaging apps. This data is encrypted between the client and server so must be intercepted inside the app's runtime before encryption. The tool should then attempt to decode and analyse the intercepted data to make it human-readable and flag up any suspected examples of private data being leaked to the app's servers.

Work Completed

A tool that intercepts TLS encryption functions at runtime, and then decodes and analyses all intercepted data. I have also created an Android app that the tool can be tested against to ensure that it is correctly intercepting and interpreting all data. I then systematically evaluated this tool on five popular messaging apps and recorded the results in chapter 4.

Special Difficulties

None

[1]This word count was computed by Microsoft Word

[2]This code line count was computed by `cloc`

Contents

List of Figures

Chapter 1

Introduction

Messaging apps are a critical part of modern life, used by over 3 billion people worldwide. Many messaging apps are advertised as 'End-to-end encrypted' (E2EE) which even privacy-conscious individuals may assume are private. During E2EE messaging, the message *contents* are encrypted so that the intermediate server is unable to read them, but accompanying *metadata* is not inside the E2EE envelope (see Figure 1.1). This metadata will be stored on the company's servers and poses a potential privacy risk for users. Once personal data has left the user's device, there is no way of knowing who may gain access to it. Possible scenarios include: it being stolen by a malicious third party that has gained access to the company servers, it being sent directly to government agencies without the user's consent [1], or the company itself using it unethically. Hagen et al. demonstrated that once a user's contacts have been uploaded to the service provider it can often be accessed by any attacker [2].

The aim of the project is to give researchers access inside of the E2EE envelope of Android devices that they control. They can then assess and compare how different messaging apps treat the user's privacy and advise for or against specific apps. A similar analysis was performed by Rottermanner et al. in 2015 [3], however, implementation details have changed significantly since then, and newer apps have since been released which have not been studied as thoroughly.

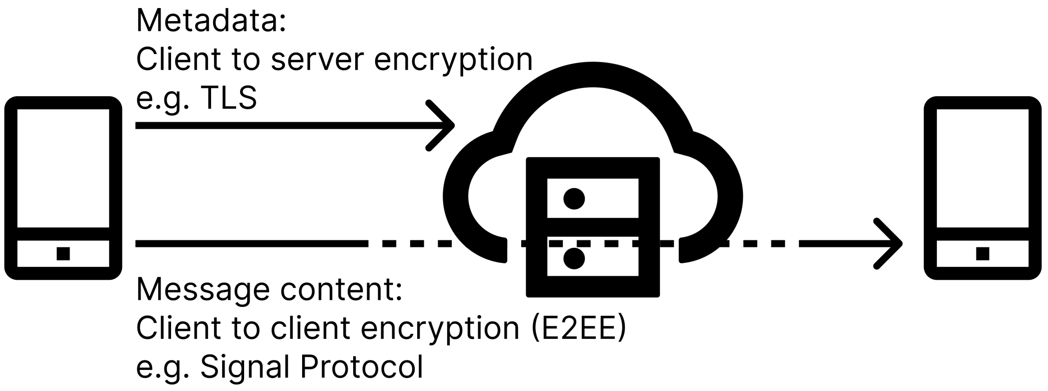

Metadata:
Client to server encryption
e.g. TLS
Message content:
Client to client encryption (E2EE)
e.g. Signal Protocol

Figure 1.1: Transport encryption vs End-to-end encryption

Chapter 2

Preparation

2.1 Potential methods

In this section I discuss some potential methods for intercepting transmitted data.

2.1.1 Packet sniffing

Given the task of intercepting transmitted metadata, a sensible first approach might be to setup a packet sniffer on the phone's local network.

Unfortunately this approach doesn't work for two reasons: All data is encrypted and it is very hard to work out which process each packet was sent from. The most common protocol used for this transport encryption is **Transport Layer Security (TLS)**.

2.1.2 Library patching

I then decided to perform the interception on the device itself. Initially I intended to do this from an Android app running on a rooted phone[1] that could overwrite system cryptographic libraries (such as OpenSSL's libssl) with patched versions that report all usage to my app. I subsequently abandoned this approach after further experimentation showed that it would be extremely challenging to work out which libraries and functions are being used by which apps. Additionally, more recent Android versions are even more restrictive with what commands apps are allowed to run, even when rooted, which made it tricky to reliably apply the required library patches.

2.1.3 Dynamic injection

Instead I used the **Frida** toolkit [4]. Frida is a dynamic code instrumentation toolkit, which allows injecting scripts into any Android app at runtime. The injected JavaScript has full access to the application's memory and can call app-defined Java functions. Using Frida, I can enumerate class and method names from the app and log whenever they are

[1]A phone on which the user has administrative (superuser) permissions, and so can modify system files

called. This helps me to work out how the app functions internally and decide where to inject my own code.

2.2 Starting point

No implementation work had been done prior to the start of this project. I had no experience with using Frida, and only a basic understanding of the structure of Android apps.

2.3 Requirements Analysis

A successful project will have met all of the following requirements:

1. A straightforward installation process that can be used across different devices and Android versions
2. Interception of transmitted data from some of the following apps:
    (a) Signal
    (b) Telegram
    (c) WhatsApp
    (d) Facebook Messenger
    (e) Wire
3. Interception of transmitted data in the following scenarios:
    (a) Account registration
    (b) Passive usage (which will still upload information such as "Last Seen" times)
    (c) Messaging one-to-one
    (d) Messaging group chats
    (e) Messaging with "Disappearing Messages" enabled[2]
4. Detailed logging of all data
5. Attempts to decode and interpret all data

2.4 Target apps

I've chosen a variety of popular messaging apps to target my analysis at. This section contains some background information about each app. TLS is a popular choice for transport encryption (client to server) and the Signal Protocol is commonly used for end-to-end encryption (client to client).

[2]"Disappearing Messages" causes messages to be automatically deleted a specified amount of time after they have been seen

	Signal	**Telegram**	**WhatsApp**	**Messenger**	**Wire**
Active users	40M [5]	500M [6]	2B [7]	1B [8]	500k [9]
Source	Open	Open	Closed	Closed	Open
Transport Encryption	TLS	MTProto [10]	Noise [11]	TLS	TLS
E2E Encryption	Signal	MTProto*	Signal [11]	Signal* [12]	Proteus†[13]

*End-to-end encryption not enabled by default †Based on Signal Protocol

2.4.1 Signal

Signal is an open source messaging app with a focus on security and privacy. It has been endorsed by privacy advocate Edward Snowden and former Twitter CEO Jack Dorsey and currently has around 40M active users [5]. All messages (in both private and group chats) are end-to-end encrypted using the Signal protocol.

2.4.2 Telegram

Telegram is open source and advertises strong privacy and security, however end-to-end encryption is not enabled by default and cannot be used with group chats. It uses a custom protocol designed for Telegram called MTProto for both transport and end-to-end encryption [10].

2.4.3 WhatsApp

WhatsApp is the most popular messaging app in the world with over 2 billion active users [7][8]. It is owned by Meta (formally Facebook) and uses the Signal Protocol to support end-to-end encryption in all private and group chats [11]. WhatsApp uses the Noise Pipe protocol to provide transport encryption instead of TLS.

2.4.4 Facebook Messenger

Messenger is also owned by Meta but is fully integrated into the Facebook platform. Users sign up using their Facebook accounts and can message any user without needing to know their phone number. Messenger also uses the Signal Protocol for end-to-end encryption in both private and group chats [12] but the encryption is not enabled by default.

2.4.5 Wire

Wire is open source and emphasises security and privacy as core values. The main product is designed for businesses, but they also offer a personal accounts which is what I will be using for analysis. It uses Proteus for end-to-end encryption which is based on the Signal Protocol [13].

Chapter 3

Implementation

3.1 Repository overview

```
TLSIntercept
├─ logs/................ Example output
│  └─ test-2022-04-21T23:49:21-org.thoughtcrime.securesms/
│     ├─ 2022-04-21T23:49:21-timeline.log
│     ├─ 2022-04-23T16:27:19.142509-144526301-sent-raw.hex
│     ├─ 2022-04-23T16:27:19.142509-144526301-sent-processed.txt
│     ├─ 2022-04-23T16:27:29.500412-180663554-received-raw.hex
│     ├─ 2022-04-23T16:27:29.845250-180663554-received-raw.hex
│     ├─ combined-180663554-received-raw.hex
│     ├─ combined-144526301-sent-raw.hex
│     └─ combined-144526301-sent-processed.txt
├─ inject.py............ Entry point
├─ process_data.py...... Decoding and searching through intercepted data
├─ conscrypt.js......... Main Frida script for intercepting TLS encryption
├─ enum.js.............. Frida script for enumerating class and method names
├─ providers.js......... Frida script for intercepting calls to Security.getProviders()
├─ signaldemo.js........ Frida script for demonstrating interception of Signal messages
├─ wordlist.txt......... Example wordlist used by process_data.py for data analysis
├─ basic/............... Frida scripts for checking basic functionality with test app
│  └─ argfunc.py  basicfunc.py  returnfunc.py
├─ test-app/............ Minimal Android app for testing TLS interception
│  └─ app/src/main/
│     ├─ java/com/interceptiontest/MainActivity.java
│     └─ proto/test.proto  Example Protocol Buffer used in MainActivity.java
├─ install.sh........... Bash script for installing Frida locally and on the device
└─ run.sh............... Bash script for starting the Frida server on the device
```

Use the following shell command to run the program

```
$ python inject.py <process> <script>
```

`<process>` can be the full name (e.g. `org.thoughtcrime.securesms`) or a shorthand (e.g. `signal`)

`<script>` defaults to `conscrypt`

3.2 Example interception

The following code illustrates the use of Frida to intercept and reimplement a Java method.

```
const PushTextSendJob = Java.use(
  'org.thoughtcrime.securesms.jobs.PushTextSendJob');
PushTextSendJob.deliver.implementation = function(record) {
  console.log('Message intercepted: ${record.getBody()}');
  return this.deliver(record)
}
```

In this situation, I've looked at Signal's source code to determine that `PushTextSendJob.deliver` is a method with one parameter which is the message that is being sent. In order to intercept the contents of the TLS envelope I traced through Signal's source code and arrived at `SSLSocket` [14]. Whilst `SSLSocket` is the documented class name, using Frida on any of its methods (e.g. `SSLSocket.connect()`) has no effect because it is only an abstract class with no actual functionality.

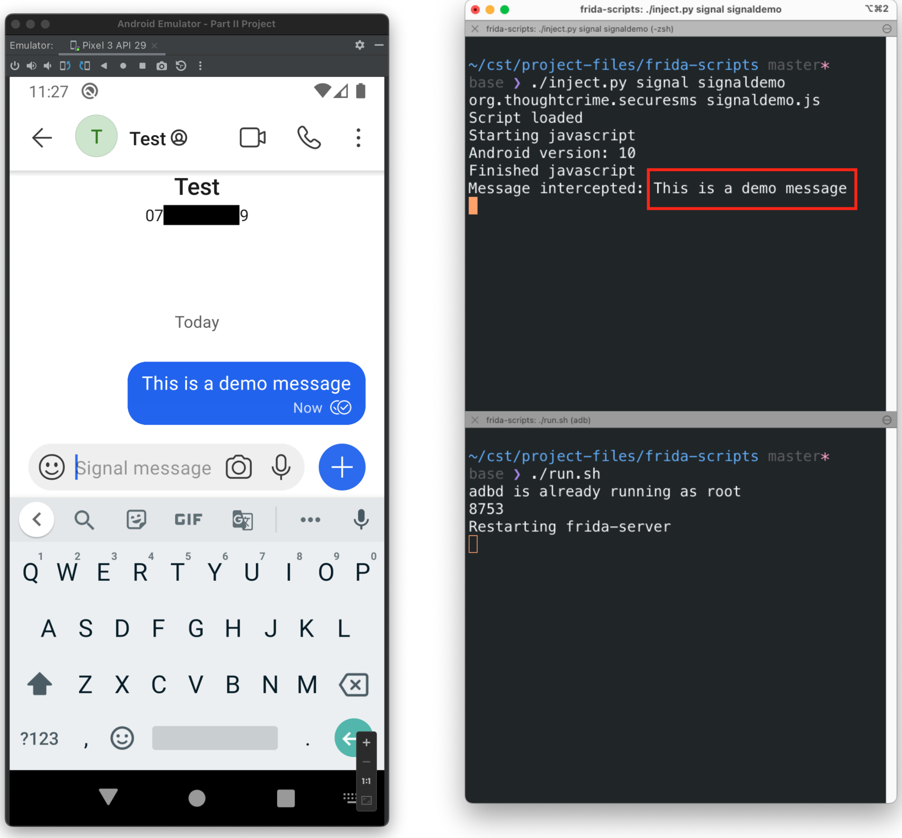
3.3 OutputStream interception

Since the intention is to create a tool that is useable with any messaging app, I began looking at libraries used for TLS. By calling `Security.getProviders()`[1] [15] I can see that Conscrypt is used to provide a TLS implementation [16]. The class `ConscryptFileDescriptorSocket` implements `SSLSocket` so now I have something that can actually be intercepted. Conscrypt is a cryptographic library by Google that is the default provider of TLS encryption for Android apps [16]. Since it is so widely used, it is hoped that a single solution will work for several of the target apps.

The socket object creates an instance of `SSLOutputStream`, which the calling program can use to send data into the socket using `SSLOutputStream.write()`. This means that the following snippet can intercept all data being sent via TLS:

```
const OutputStream = Java.use(
  'org.conscrypt.ConscryptFileDescriptorSocket$SSLOutputStream');
OutputStream.write.overload('[B', 'int', 'int').implementation =
function(byteArray, offset, byteCount) {
  processData(byteArray, offset, byteCount, this.hashCode());
  this.write(byteArray, offset, byteCount);
}
```

[1] As implemented in `providers.js`



```
frida-scripts: ./inject.py signal signaldemo
frida-scripts: ./inject.py signal signaldemo (-zsh)

~/cst/project-files/frida-scripts master*
base ❯ ./inject.py signal signaldemo
org.thoughtcrime.securesms signaldemo.js
Script loaded
Starting javascript
Android version: 10
Finished javascript
Message intercepted: This is a demo message
```

```
frida-scripts: ./run.sh (adb)

~/cst/project-files/frida-scripts master*
base ❯ ./run.sh
adbd is already running as root
8753
Restarting frida-server
```

(a) Phone (b) Terminal

Figure 3.1: Screenshot demonstrating Signal code being intercepted

Received data can be intercepted using the analogous `SSLInputStream.read()`.

Some large HTTPS requests are split across multiple `SSLOutputStream.write()` operations so `SSLOutputStream.hashCode()` provides a unique ID for each object. This allows them to be recombined later into a single file.

3.4 Other methods

Unfortunately, this approach does not work for all targeted apps, in particular WhatsApp and Telegram.

3.4.1 WhatsApp

WhatsApp uses Noise Pipes from the Noise Protocol Framework [11] for transport security instead of the TLS protocol. This means that all encryption functionality is performed inside the app instead of using Conscrypt. I have used a decompilation tool [17] to decompile the WhatsApp APK in order to find the encryption function for interception.

The decompilation reveals that most token names have been obfuscated: the following is an excerpt from the decompilation results:

```
if (!C62802ra.A00().A02(r7.A01, A014, r22.A02.A01())) {
    str2 = "invalid signature on noise certificate";
} else if (!Arrays.equals(r5.A03.A01(), r4.A01)) {
    str2 =
      "noise certificate key does not match proposed server static key";
}
```

This means that intercepting WhatsApp's encryption functionality is not possible without a lengthy deobfuscation attempt, which I have not performed.

3.4.2 Telegram

Telegram uses a custom protocol called MTProto for its transport security. Since it is open source, I have investigated and discovered that all of its important functionality is implemented in C++. Frida is unable to intercept C++ functions using the same methods as Java, so I have been unable to intercept Telegram's encryption.

3.5 Data processing

The intercepted data is in the form of an array of bytes with no additional context. This is logged in its raw form and separately processed and decoded. Some requests are split across multiple calls to `SSLOutputStream.write()` so every write is processed both individually and after concatenation with all previous writes to the same `SSLOutputStream` object.

Some intercepted data includes extra bytes at the beginning which are not part of the actual data structure. These extra bytes may not be valid in the encoding used by the rest of the transmission (usually UTF-8). Because of this, decoding is attempted multiple times with incremented start offsets until a valid start offset is found.

3.5.1 Decoding

The intercepted data comes in a variety of formats and encodings. The decoding process attempts various different methods and keeps the first successful one. It uses Python's `json` library for JSON decoding and `base64` for decoding Base64. Some JSON values are encoded with Base64 so Base64 decoding is also attempted on all JSON values.

Protocol buffers (ProtoBufs) are a common alternative to JSON for serialising structured data [18]. ProtoBufs appear as binary data with no apparent structure, because to interpret them fully you need the original definition. However, this tool makes use of the `blackboxprotobuf` library [19] which can decode a ProtoBuf without its definition. This process loses the field names, so future development could consider extracting all ProtoBuf definitions from the app APK and testing each of them against the discovered ProtoBuf to see which would be valid decodings.

Example ProtoBuf definition

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

Example blackboxprotobuf output

```
{
  "1": "Test Name",
  "2": "123456",
  "3": "test@example.com"
}
{
  '1': {'type': 'bytes', 'name': ''},
  '2': {'type': 'int', 'name': ''},
  '3': {'type': 'bytes', 'name': ''}
}
```

3.5.2 Analysis

For data analysis I provide a customisable text file (`wordlist.txt`) that is loaded into the program. All listed words are searched for in the decoded data, and any matches are notified in the log file under `DATA_ALERTS`. Examples of words in the list include "location", "address", "contact", and "image".

3.6 Test app

In order to be sure that both the interception and decoding works correctly, I have made a minimal Android app for testing.

Java code to generate a GET request

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

Java code to send data over an SSL Socket

```
Socket socket = factory.createSocket("example.com", 443);
OutputStream outputStream = socket.getOutputStream();
outputStream.write("{\"key\": \"val\", \"location\": \"UK\".getBytes());
```

Java code to build and send a ProtoBuf

```
TestProtoBuf.Builder builder = TestProtoBuf.newBuilder();
builder.setName("Test Name");
builder.setId(123456);
builder.setEmail("test@example.com");
builder.setType(TestProtoBuf.TestEnum.OPT2);
TestProtoBuf protoBuf = builder.build();
protoBuf.writeTo(outputStream);
```

This demonstrates that the TLS interception works reliably, the JSON and ProtoBuf decoders work, and the string `"location"` is correctly detected and flagged.

Chapter 4

Evaluation

For evaluation, the following scenarios will be executed on all target apps whilst the interception is running.

(a) Account registration

(b) Passive usage (which will still upload information such as “Last Seen” times)

(c) Messaging one-to-one

(d) Messaging group chats

(e) Messaging with “Disappearing Messages” enabled[1]

Testing is being performed on a Google Pixel 3 running Android 12[2], using a completely new phone number (07375810067). Two other phone numbers have been inserted into the phone’s contacts list and the contacts’ names and numbers have been added to the wordlist from subsection 3.5.2.

4.1 Signal

Overall, the tool worked extremely well with Signal. It intercepted all connections and decoded many of them into easily understood plaintext.

4.1.1 Account registration

During registration, Signal asks the user for permission to access their contact list and phone calls. It then validates the SMS that the user is signing up with. The HTTPS requests are successfully intercepted and readable as plain text. No phone numbers of contacts were detected as plain text.

[1]“Disappearing Messages” causes messages to be automatically deleted a specified amount of time after they have been seen

[2]October 2021 update, build number SP1A.210812.016.C1

Request for SMS authentication code

```
GET /v1/accounts/sms/code/+447375810067?client=android-2021-03&
  challenge=3e9fa22f74f17160cba70eb8090ddf5f HTTP/1.1
Authorization: Basic KzQ0NzM3NTgxMDA2NzpFdzBnVFBTaUhGMlIvVkR1U0E1Vkpxb1Y=
X-Signal-Agent: OWA
User-Agent: Signal-Android/5.36.3 Android/31
Host: chat.signal.org
Connection: Keep-Alive
Accept-Encoding: gzip
```

Transmitted device information

```
{
  'capabilities':
  {
    'announcementGroup': True, 'changeNumber': True, 'senderKey': True,
    'storage': True, 'stories': False, 'uuid': False,
  },
  'discoverableByPhoneNumber': True, 'fetchesMessages': False,
  'name': None, 'pin': None, 'registrationId': 1834,
  'registrationLock': None, 'signalingKey': None,
  'unidentifiedAccessKey': '2Eun0A0k1BSfdDqlPIopeA==',
  'unrestrictedUnidentifiedAccess': False,
  'video': True, 'voice': True
}
```

4.1.2 Passive usage

Every time the app is started it opens a websocket connection with Signal's servers:

Signal websocket authentication request (excerpt)

```
GET /v1/websocket/?
  login=17a3e26b-0f8a-4bbd-9833-b694ac3425c0&
  password=zTDykT85PUtwy2+3wB4fkAol HTTP/1.1
```

The following request is sometimes observed:

(excerpt)

```
PUT /v1/accounts/gcm/ HTTP/1.1
Authorization: Basic MTdhM2UyNmItMGY4YS00YmJkLTk4MzMtYjY5[...]A==
X-Signal-Agent: OWA
User-Agent: Signal-Android/5.36.3 Android/31
Content-Type: application/json; charset=utf-8
Content-Length: 211
Host: chat.signal.org
Connection: Keep-Alive
Accept-Encoding: gzip
```

```
{"gcmRegistrationId":"cdb-gO45QUKP3sweTkSK2N:APA91bFOmPId[...]NT__pLob",
"webSocketChannel":true}
```

4.1.3 Messaging one-to-one

When a message is sent, some binary data is intercepted but it isn't decoded because I'm not sure what format it is in. I wouldn't expect all the data in these packets to be decodable as some of it will be the message contents after E2EE.

4.1.4 Messaging group chats

Some API requests are made to `/v1/groups/` but nothing of interest is observed.

4.1.5 Messaging with "Disappearing Messages" enabled

Only undecodable binary data is observed.

4.2 Telegram

As mentioned in subsection 3.4.2, Telegram implements all transport security in C++. This means that no data is expected to be intercepted in any of the tests.

In testing, this was mostly true, with the exception of the first opening of the app:

```
POST /v1/projects/tmessages2/installations HTTP/1.1
Content-Type: application/json
Accept: application/json
Content-Encoding: gzip
Cache-Control: no-cache
X-Android-Package: org.telegram.messenger
x-firebase-client: android-target-sdk/30 device-model/blueline fire-core/
  20.0.0 device-name/blueline android-min-sdk/23 fire-abt/
  21.0.0 android-installer/com.android.vending fire-installations/
  17.0.0 fire-android/31 device-brand/google fire-rc/21.0.1 fire-fcm/
  22.0.0 android-platform/
x-firebase-client-log-type: 3
X-Android-Cert: 9723E5838612E9C7C08CA2C6573B6026D7A51F8F
x-goog-api-key: AIzaSyA-t0jLPjUt2FxrA8VPK2EiYHcYcboIR6k
User-Agent: Dalvik/2.1.0 (Linux; U; Android 12;
  Pixel 3 Build/SP1A.210812.016.C1)
Host: firebaseinstallations.googleapis.com
Connection: Keep-Alive
Accept-Encoding: gzip
Content-Length: 134
```

This demonstrates that Telegram are collecting analytics data on all installations, including the exact build number of the phone's operating system.

4.3 WhatsApp

Once again, I wasn't expecting to intercept any data (due to reasons discussed in subsection 3.4.2). However several GET requests were intercepted immediately after opening the app for the first time, including the following:

```
GET /generate_204 HTTP/1.1
User-Agent: Dalvik/2.1.0
  (Linux; U; Android 12; Pixel 3 Build/SP1A.210812.016.C1)
Host: clients3.google.com
Connection: Keep-Alive
Accept-Encoding: gzip
```

This demonstrates that WhatsApp are also collecting the OS build number. The other GET requests were for downloading stickers and wallpapers.

4.4 Messenger

Despite Messenger apparently using TLS for all client-server communications, during evaluation no data was intercepted during normal operation.

However, I observed an internal crash that occurs shortly after loading the app. This crash is handled by the app and isn't observable to the user, but a very large crash log (600KB of text) is uploaded via HTTPS. The log contains data such as the amount of memory on the device, times of recent crashes, whether the device is 'jailbroken', how much battery it has left, and the entire info log including stack traces. This large amount of data could certainly be used to fingerprint a device across different apps.

Excerpts from Messenger's uploaded crash log

```
POST /mobile/reliability_event_log_upload/ HTTP/1.1
User-Agent: Android
Content-Type: multipart/form-data;
  boundary=738f3379-20b5-4172-ac5c-dfcddefb0440
Cookie: c_user=100078408014720
Transfer-Encoding: chunked
Host: b-www.facebook.com
Connection: Keep-Alive
Accept-Encoding: gzip

Content-Disposition: form-data; name="log_type"
android_large_unexplained

Content-Disposition: form-data; name="data[available_memory]"
```

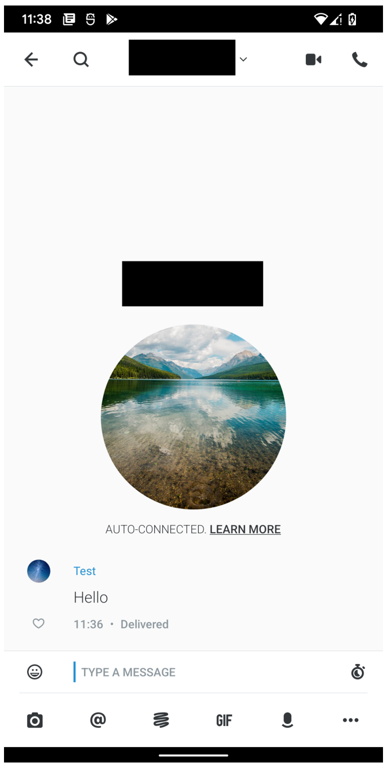

11:38
[redacted]
[redacted]
AUTO-CONNECTED. LEARN MORE
Test
Hello
11:36 · Delivered
TYPE A MESSAGE

Figure 4.1: Message being sent using Wire

```
3075735552

Content-Disposition: form-data; name="data[battery_energy_counter]"
-9223372036854775808

Content-Disposition: form-data; name="data[is_employee]"
false
```

4.5 Wire

With Wire, all transmissions were easily viewable. The only thing found of interest is that the user's password is not hashed before transmission.

```
POST /activate/send HTTP/1.1
Content-Type: application/json
Content-Length: 42
Host: prod-nginz-https.wire.com
Connection: Keep-Alive
Accept-Encoding: gzip
User-Agent: okhttp/3.14.9

{"phone":"+447375810067","locale":"en-GB"}
```

```
{"name":"Test",
"locale":"en-GB",
"email":"tpmb4@cam.ac.uk",
"password":"my$ecretPassword",
"email_code":"260827",
"label":"b103f9bb-1efa-4a4f-ac4f-c6114bcd10ef"}
```

Chapter 5

Conclusion

The aim of giving researchers access inside the E2EE envelope of Android messengers has been fully achieved for some of the target apps (Signal and Wire) and partially achieved for the other three apps (WhatsApp, Messenger, and Telegram). I think that I have approached the limit of what is possible using the dynamic injection technique and further improvements would likely require new interception methods.

The automated decoding and analysis of intercepted data has also been successful, although many sections of data have not been decoded to a readable format. It is impossible to know how much of this is theoretically decodable since much of it is E2EE data that the server cannot decrypt either.

I have not found any major violations of privacy in any of the targeted apps, although the intercepted Facebook Messenger crash logs contain data which is likely sufficient to fingerprint the device. It is possible that more serious privacy leakages can be found with better decoding techniques.

Finally, this tool has already proved useful for security research outside of privacy concerns: the interception of Signal authentication credentials has been a crucial part of an active vulnerability disclosure being worked on by myself with other security researchers.

Bibliography

[1] "Nsa prism program taps in to user data of apple, google and others," June 2013, last accessed 12 May 2022. [Online]. Available: https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data

[2] C. Hagen, C. Weinert, C. Sendner, A. Dmitrienko, and T. Schneider, "All the numbers are us: Large-scale abuse of contact discovery in mobile messengers," Cryptology ePrint Archive, Report 2020/1119, 2020, https://ia.cr/2020/1119.

[3] C. Rottermanner, P. Kieseberg, M. Donko-Huber, M. Schmiedecker, and S. Schrittwieser, "Privacy and data protection in smartphone messengers," 12 2015, pp. 1–10.

[4] "Frida homepage," last accessed 12 May 2022. [Online]. Available: https://frida.re/

[5] "Signal revenue & usage statistics," 2022, last accessed 12 May 2022. [Online]. Available: https://www.businessofapps.com/data/signal-statistics/

[6] "Telegram revenue and usage statistics," 2022, last accessed 12 May 2022. [Online]. Available: https://www.businessofapps.com/data/telegram-statistics/

[7] "Two billion users – connecting the world privately," 2020, last accessed 12 May 2022. [Online]. Available: https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately

[8] "Most popular global mobile messenger apps," 2022, last accessed 12 May 2022. [Online]. Available: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

[9] "Wire messenger review," 2021, last accessed 12 May 2022. [Online]. Available: https://securitytech.org/secure-encrypted-messaging-app/wire/

[10] "Telegram mtproto documentation," last accessed 12 May 2022. [Online]. Available: https://core.telegram.org/mtproto

[11] "Whatsapp encryption overview – technical white paper," November 2021, last accessed 12 May 2022. [Online]. Available: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

[12] "Messenger secret conversations – technical whitepaper," May 2017, last accessed 12 May 2022. [Online]. Available: https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf

[13] "Wire security whitepaper," July 2021, last accessed 12 May 2022. [Online]. Available: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf

[14] "Android documentation: Sslsocket," last accessed 12 May 2022. [Online]. Available: https://developer.android.com/reference/javax/net/ssl/SSLSocket

[15] "Android documentation: Security.getproviders()," last accessed 12 May 2022. [Online]. Available: https://developer.android.com/reference/java/security/Security/#getProviders()

[16] "Conscrypt homepage," last accessed 12 May 2022. [Online]. Available: https://conscrypt.org/

[17] "Decompiler.com," last accessed 12 May 2022. [Online]. Available: https://decompiler.com/

[18] "Protocol buffers documentation," last accessed 12 May 2022. [Online]. Available: https://developers.google.com/protocol-buffers

[19] "Blackbox protobuf library," last accessed 12 May 2022. [Online]. Available: https://github.com/ydkhatri/blackboxprotobuf

Detecting privacy leaks in Android messengers

Part II Project Proposal

1 Introduction

When using a smartphone messaging app (such as WhatsApp), metadata is sent to the server alongside the content of your messages. Even if the messages are end-to-end encrypted, much of the metadata is not. The unencrypted metadata sometimes includes personal information (such as the user's location, list of contacts, message burn time, and image thumbnails), or information useful for fingerprinting devices (such as screen resolution or exact operating system version).

Because the personal information can be read by anyone with access to the messaging servers, it may constitute a violation of the user's privacy. The reasons that this data is being sent vary: it may be necessary for functionality, accidentally included, being used to build up advertising profiles, or deliberately added maliciously. For an example of the latter, a secure chat app called EncroChat was infiltrated by police. They published an updated version of the app which transmitted users' location alongside each message, violating the privacy of the app's users without them knowing.

A standard user has no way of knowing what data is being leaked when they use a service, and it is difficult even for security researchers to analyse. This project aims to build a tool that will extract metadata being sent by a device and flag up potential privacy violations.

2 Starting point

Some similar research was done in *Privacy and Data Protection in Smartphone Messengers* (2015), which I shall using for reference during my evaluation. All other research has been from discussions with my supervisors and is contained within this document. No implementation work has been done prior to the start of this project. I have some programming experience in Java, some theoretical knowledge of TLS, and a basic understanding of how to create an Android app.

3 Description

The objective is to create an Android app that runs as root. This app will list all other installed apps and allow the user to select which ones should be analysed. During operation of the target apps, all internet traffic will be logged. The app will allow exporting the data in its 'raw' form and will process it so that all of the data can be displayed in a human-readable format. It will automatically analyse the data to flag up potentially interesting information.

Whilst the data of interest is not end-to-end encrypted, it is likely to be SSL/TLS encrypted during transfer. This renders packet-sniffing useless unless you have access to the SSL certificate private keys of the destination servers. Since the intention is to observe the target apps without changing their behaviour, we have no way of getting around this limitation for packet-sniffers. Instead, my app will overwrite the system TLS libraries and modify them so that my app can intercept data being TLS-encrypted and store it for analysis. This limits the effectiveness of this tool to only work with apps that use system TLS libraries. Any apps that bundle their own TLS libraries will have to be dealt with individually and are outside the scope of this project (but see Potential Extensions below).

I am expecting that the metadata will be transmitted as JSON, XML, or Protocol Buffers. JSON and XML should be straightforward to process because the field names are included. Since Protocol Buffers don't transmit the field names, they will be harder to process, but any UTF-8 text can still be detected. For all 3 formats I will be using a combination of generic heuristics and app-specific processing based on manual inspection of that app's metadata.

The automatic processing will include techniques including attempting different decodings, searching for human-readable text, and using wordlists to detect keys of interest (such as "location" or "contacts").

Most development will be done using an Android emulator, but I will be regularly testing the app on a variety of real devices.

3.1 Potential extensions

1. Investigate and implement TLS library patching for apps that don't use system TLS libraries.

2. Enumerate required permissions that could leak private information for each target app.

3. Apply some information theory concepts, such as entropy, to the captured metadata to improve detection of interesting data.

4. Investigate if similar capturing techniques would work with other categories of apps beside messaging, or the entire operating system.

5. Much of the data leaving the device will be end-to-end encrypted. This means that only the destination device will be able to read this data and not the intermediate server, so it is of much less interest when analysing information leakage.

   Despite this, investigating various aspects of E2EE could provide an interesting extension to the project:

   a) Explore patching one or two specific apps to break their AES encryption, possibly by changing their random number generators to be no longer cryptographically secure. This would allow analysis of the E2EE data.
   b) Search for symmetric keys used to encrypt E2EE messages within the databases of the target app, and then extract these and decrypt the E2EE payload for one or two specific apps.
   c) Analysing E2EE data is only uninteresting if the E2EE is actually fully secure. My app could attempt to detect whether the target apps have had their E2EE compromised.

4 Success criteria

The evaluation will be performed on real devices against a variety of messaging apps (Facebook Messenger, WhatsApp and Signal, at a minimum). These apps will be used in the following scenarios:

- Account registration
- Passive usage (which will still upload information such as "Last Seen" times)
- Messaging one-to-one
- Messaging group chats
- Messaging with "Disappearing Messages" enabled

A successful project will be able to intercept all non-E2EE data being sent by the target apps during the above scenarios and display it to the user, even if the data was TLS-encrypted. If the selected target apps are not using system TLS libraries, then I will implement extension 1 or substitute other messaging apps such as Telegram, Line, and WeChat.

It will be able to decode the metadata formats used by the target apps and display the data in a more human-readable format. Using this, it will then be able to detect privacy-violating data leaving the device (such as the user's exact OS version, location or a list of contacts) and highlight these for the user.

Since Signal is open source, I can inspect its source code manually to find out what data is actually sent during the above scenarios. During evaluation I will compare these findings with the report generated by my app. Even though the other target apps are closed source, I will still be expecting similar types of data to be found in the generated reports, and any additional interesting or unexpected findings will be further evidence of the project's success.

5 Timetable

	Due date	Tasks
1	29 October	Research Android rooting process; Create basic app with root access
2	12 November	Investigate system TLS libraries
3	26 November	Implement TLS library patch that sends data to app
4	10 December	Implement data processing and exporting code and UI
5	7 January	Implement data analysis heuristics
6	21 January	Write progress report and presentation; Start evaluation using at least 1 messaging app, improving data processing and analysis heuristics where necessary
7	11 February	Continue evaluation using at least 1 other messaging app; Start dissertation: Introduction draft
8	25 February	Continue evaluation; Continue dissertation: Preparation draft; Implementation started
9	18 March	Continue dissertation: Complete draft
10	8 April	Continue dissertation: Second draft
11	29 April	Finish dissertation: Final draft and submission

The most unpredictable part of the project is the TLS library patching, which has been placed early in the project timetable (sections 2 & 3). If this goes well, then extensions 1–3 may be considered and implemented during this time. Extension 4 will be considered during evaluation (sections 6–8).

6 Resource declaration

Development will happen on my personal laptop: 2.3GHz 8-Core Intel Core i9, 16GB RAM, 1TB SSD, macOS. The entire filesystem is automatically backed up to an external hard drive every day. I will be using git for both the project and the dissertation and will push daily to a remote repository on GitHub. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Development and testing will primarily be on an emulated Android device but will also occur on physical devices. I accept responsibility for obtaining these, either independently or from the department. These will be useful for the research, but no single device is critical.