



**Trinity College Dublin**

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

School of Computer Science and Statistics

# How Private Are Android Keyboards?

Kamil Przepiorowski

*Supervised by Douglas Leith*

April 18, 2022

A Dissertation presented to the  
University of Dublin, Trinity College,  
in partial fulfilment of the requirements for  
the degree of Masters of Computer Science (MCS)

# Declaration

I, Kamil Przepiorowski, hereby declare that this Dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at <http://www.tcd.ie/calendar>.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at <http://tcd-ie.libguides.com/plagiarism/ready-steady-write>.

Signed: \_\_\_\_\_

Date: \_\_\_\_\_

# Abstract

By their very nature, keyboards process everything that users input into their devices, including private messages, passwords and payment details. Despite this, their privacy is often overlooked, and in fact, not much is understood about the privacy considerations of even the most popular keyboards available on Android. This dissertation addresses this issue and investigates the data collected and shared with back-end servers by some of the most popular keyboard applications, namely Google's Gboard, Microsoft's SwiftKey and an open-source alternative, AnySoftKeyboard. Providing users with a deeper understanding of the underlying privacy considerations of these keyboards will allow them to make more informed decisions about their consent to the ongoing data collection.

The privacy investigation is performed by collecting the network traffic generated by each keyboard, followed by a significant amount of reverse engineering work required to decrypt and decode the data, whose content is then analysed for the presence of sensitive information.

It is found that the two proprietary keyboard applications both collect and share large amounts of telemetry data. Gboard provides an opt-out from this data collection; however, SwiftKey does not. The data logged by these keyboards is largely similar. It includes timestamped log entries containing specific hardware information, the length of individual words entered, languages used, and the application's name in which the keyboard was opened. Neither keyboard has been observed to collect or share the input content or even track the frequency of individual characters. Most notably, the logs of both keyboards include unique identifiers, namely the Android ID and the Google Advertising ID, corresponding to Gboard's and SwiftKey's logs, respectively. This allows the data to be linked to a specific handset and potentially a user's identity, putting their anonymity at risk. Additionally, it is possible to infer certain personality traits about users based on the timestamped application usage information found in both keyboards' logs. It is also found that the open-source AnySoftKeyboard lives up to its reputation as a privacy respecting keyboard. It is not observed to collect or share any telemetry at all, proving that excessive telemetry collection is indeed a choice made by developers and not a requirement.

# Acknowledgments

I firstly want to thank my supervisor, Professor Douglas Leith, for his continuous support, feedback and guidance throughout the duration of this project.

I would also like to extend my gratitude to my friends, and my loving partner, whose constant support and encouragement have been invaluable during my five years at Trinity.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgments</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Research Objectives . . . . .	2
1.3 Dissertation Structure . . . . .	3
<b>2 Background</b>	<b>4</b>
2.1 User Privacy . . . . .	4
2.1.1 Sensitive Data . . . . .	4
2.1.2 Threat Model . . . . .	4
2.2 Security . . . . .	5
2.2.1 Man in the Middle . . . . .	5
2.2.2 Certificate Pinning . . . . .	5
2.3 Data Formats . . . . .	6
2.3.1 Protocol Buffers . . . . .	6
2.3.2 Avro . . . . .	7
2.4 Android . . . . .	7
2.4.1 Rooting . . . . .	7
2.4.2 APK . . . . .	7
2.5 Tools . . . . .	7
2.5.1 mitmproxy . . . . .	7
2.5.2 jadx . . . . .	8
2.5.3 Frida . . . . .	8
2.5.4 Android Debug Bridge . . . . .	8
<b>3 Related Work</b>	<b>9</b>
<b>4 Experimental Setup</b>	<b>12</b>

4.1	Data Collection . . . . .	12
4.2	Hardware and Software Used . . . . .	13
4.3	Experiments . . . . .	14
<b>5</b>	<b>Reverse Engineering</b>	<b>17</b>
5.1	Gboard . . . . .	18
5.2	SwiftKey . . . . .	25
<b>6</b>	<b>Gboard</b>	<b>29</b>
6.1	Application Overview . . . . .	29
6.2	Permissions . . . . .	29
6.3	Privacy Policy . . . . .	30
6.4	Opting Out . . . . .	31
6.5	Experiment Results . . . . .	32
<b>7</b>	<b>SwiftKey</b>	<b>38</b>
7.1	Application Overview . . . . .	38
7.2	Permissions . . . . .	38
7.3	Privacy Policy . . . . .	38
7.4	Opting Out . . . . .	40
7.5	Experiment Results . . . . .	41
<b>8</b>	<b>AnySoftKeyboard</b>	<b>45</b>
8.1	Application Overview . . . . .	45
8.2	Permissions . . . . .	45
8.3	Privacy Policy . . . . .	46
8.4	Opting out . . . . .	46
8.5	Experiment Results . . . . .	46
<b>9</b>	<b>Evaluation</b>	<b>47</b>
9.1	Permissions . . . . .	47
9.2	Transparency . . . . .	48
9.3	Data Collection . . . . .	48
<b>10</b>	<b>Conclusion</b>	<b>52</b>
10.1	Overview . . . . .	52
10.2	Future Work . . . . .	53

# List of Figures

2.1	The steps carried out in a Man in the Middle attack. . . . .	6
4.1	The experimental setup used for data collection. . . . .	13
5.1	Connections to the play.googleapis.com/log/batch endpoint. . . . .	18
5.2	The content of a log batch connection. . . . .	19
5.3	Obfuscated constructor of a Clearcut Logger class responsible for building log events. . . . .	20
5.4	Side-by-side comparison of an encoded log entry and its decoded meaning. .	23
5.5	Connections observed on startup of the SwiftKey application. . . . .	25
5.6	The experimental setup used to bypass SwiftKey's certificate pinning. . . . .	27
5.7	A summary of the decoded logs sent by SwiftKey to the telemetry endpoint. .	28
6.1	The advanced settings available on Gboard. . . . .	31
7.1	The telemetry options available on SwiftKey. . . . .	40
9.1	Example of the timestamp information of a WhatsApp conversation between two handsets, with the timestamps scaled up to seconds from milliseconds, and offset so that the conversation starts at timestamp 0. . . . .	51

# List of Tables

5.1	The field entries found within the compact protobuf definition of the log entry root protobuf. . . . .	21
5.2	The Keyboard Events present in the log entries from the LATIN_IME source.	24
6.1	The list of permissions requested by Gboard. . . . .	30
7.1	The list of permissions requested by SwiftKey. . . . .	39
8.1	The list of permissions requested by AnySoftKeyboard. . . . .	45



# 1 Introduction

## 1.1 Motivation

Android is the most popular operating system for mobile devices, and Google recently announced in their 2021 I/O conference (20) that there are now over three billion active Android devices in the world. Users must often entrust several vendors to handle their personal information when using these devices. This includes not only the Operating System (OS) developer (39), in this case, Google, but also the developers of the applications they interact with. Users are not always given a straightforward way to opt out of the collection of telemetry data when using applications and are seldom transparently told what data is collected about them, what purpose it is used for or where it is sent (47). This naturally raises concerns about how much users should really trust these companies with their personal information.

Due to this lack of transparency, and therefore lack of awareness from users, they often fall victim to what is labelled as the privacy paradox (11). Although users claim to be concerned about their privacy, they do not undertake any measures to protect their personal information. A significant contributor to this problem is the fact that opt-out options are typically hidden behind several screens within the settings, and the average user does not go through the effort to navigate them. As such, a user's decision to consent to data collection is not always explicit or informed. If users had a deeper understanding of what is tracked about them, they would be allowed to make more appropriate decisions about whether they agree to this data being collected and shared.

Keyboard applications are a core component of mobile devices. All user input is entered directly through them, including personal information, such as private messages, emails, passwords and payment details. Nevertheless, their privacy implications are not well understood, and it is unknown what data is collected and shared about users and their inputs as they interact with these applications. The collection of data about user inputs has the potential to be extremely dangerous. It may allow specific messages to be linked to a user's identity or expose sensitive credentials and personal information. Despite this, users are often unaware of what data is tracked about them (30). Therefore, they are not able to make an informed decision about their consent to the data collection performed by their keyboards.

## 1.2 Research Objectives

With the above context and motivation in mind, the primary aim of this research is to investigate the privacy implications of popular Android keyboards and determine whether they share any sensitive data with their back-end servers. Having a deeper understanding of this will allow users to truly perform an informed decision about their consent to the data collection performed by these applications.

By focusing on the most popular keyboards, this work will be able to impact the highest amount of users. As such, two of the most popular proprietary Android keyboards were chosen; namely, Google's Gboard, which has over five billion installs (19), and Microsoft's SwiftKey, with over one billion installs (43). An open-source alternative, AnySoftKeyboard, which is widely regarded as one of the best privacy oriented keyboards available, is also investigated and used as a baseline for comparison.

This report is a technical one and will discuss its findings through the lens of user privacy. It will not comment on the legality of the data collection or its compliance with laws such as the General Data Protection Regulation (GDPR).

In order to accomplish the stated research aim, the following objectives must be achieved:

- Collect data about the network traffic sent to back-end servers by the aforementioned keyboard applications.
- Decrypt and decode the captured data to a human-readable format.
- Analyse the decoded logs for any sensitive user data.
- Provide a critique of whether the data logged by each keyboard includes sensitive information and comment on whether there are opt-out options available and if users are transparently informed about the data collection in relevant privacy policies.

Achieving these objectives is not expected to be straightforward due to several obstacles. Firstly, proprietary applications are closed-source, meaning their code is not publicly available. It is possible to decompile these applications through existing software, such as jadx (50); however, the resulting code is almost always intentionally obfuscated and not human-readable (27). Secondly, the network traffic sent out by the applications is not only encrypted, but also often encoded in a binary format for which there is no public documentation available. These challenges mean that a significant amount of reverse engineering work must be done for each application before the logged data reaches a human-readable format and can be analysed and critiqued. There are also no automated tools to do this, and thus, the work must be performed manually.

## 1.3 Dissertation Structure

- **Chapter 2 - Background**, introduces core concepts and theory required to fully understand the rest of the report.
- **Chapter 3 - Related Work**, discusses existing work which has been done in the field of the privacy of Android applications.
- **Chapter 4 - Experimental Setup**, outlines the procedure and setup used for data collection.
- **Chapter 5 - Reverse Engineering**, describes in detail the process of decrypting and decoding the captured network traffic.
- **Chapters 6, 7 and 8**, discuss the **Gboard**, **SwiftKey** and **AnySoftKeyboard** applications respectively. Each chapter discusses its keyboard separately, including an overview of the requested permissions, the privacy policy, and any opt-out options available. The results of the experiments performed on each keyboard are also presented.
- **Chapter 9 - Evaluation**, provides a critical analysis of the obtained results, as well as a discussion about system permissions and transparency of all three applications.
- **Chapter 10 - Conclusion**, comments on the achievements of this research, and proposes future work.

## 2 Background

### 2.1 User Privacy

#### 2.1.1 Sensitive Data

In the context of this work, sensitive data will be defined as data that can either directly or indirectly identify or be linked to a user's real identity. This includes values such as unique identifiers or combinations of data, which on their own may be anonymous, but, when combined, can become revealing.

#### 2.1.2 Threat Model

Most applications require some knowledge about the device they are installed on and specific system permissions to function appropriately. However, it is often the case that users are not well-informed about what the application uses its permissions for and are often given little control over which permissions they can turn off. For example, it is rarely possible to deny an application network permissions if they are requested, and users are often left in the dark as to what is done with the network access. One of the primary uses of internet access is for applications to communicate with and send telemetry data to remote servers. The collection of telemetry data and its transmission to back-end servers is not intrinsically a breach of user privacy. Indeed, occasional data transmission is expected for various purposes, such as checking for updates or sending crash logs. It is often beneficial to collect some level of telemetry to help make applications more stable across different hardware and detect bugs based on crash analytics or performance statistics. If this data is not uniquely identifying and is common to a high number of devices, it does not pose substantial privacy concerns (52). The issues begin to arise when the collected data puts the users' anonymity and privacy at risk. This is often a result of application vendors collecting sensitive information excessively, both in terms of quantity and specificity. Data that is not sensitive in isolation but which can either be used to link other data together or reveal more context about it can also often be de-anonymised (56). This can lead to behavioural and usage patterns being revealed about users, potentially leading to the reveal of personality traits. For example, observing a user's

app usage over time might show that they use specific applications at certain times of the day or month, which could potentially reveal traits such as sex, gender or religious beliefs. Users are almost always unaware of what data is tracked about them due to the lack of documentation and transparent communication from application vendors (47). This research aims to shed light on this issue and bring clarity to users about what Android keyboard applications are sending to their back-end servers and whether the user's privacy is preserved or not.

## **2.2 Security**

### **2.2.1 Man in the Middle**

HTTPS connections are secured through the use of TLS certificates. When a client application receives a certificate from a server, as part of the TLS handshake, it will verify the certificate's integrity against a pre-defined list of trusted Certificate Authorities (CAs) and proceed with the connection only if this check succeeds.

A Man in the Middle (MITM) attack relies on an intermediary between a client and a server attempting to communicate. The intermediary pretends to be the destination server and intercepts all the exchanges between the two parties before forwarding them to the actual destination. If the attack succeeds, neither party is aware of the attacker and trusts that the messages are being delivered directly to their intended target.

In order to execute a MITM attack, the TLS handshake process must be hijacked. When a client initiates a connection with a server, the server will respond with a TLS certificate signed by a certificate authority. The MITM intercepts this and instead delivers its own self-signed certificate to the client. If the client trusts this certificate, it will accept it and encrypt its network data with the MITM's public key, which can then be decrypted at the intermediary. At this point, the content of the message can be read, logged, and potentially altered. The MITM then re-encrypts the data with the destination server's public key and forwards the message accordingly. This makes the MITM appear transparent to both the client and the server, who both believe they are in a secure exchange. The man in the middle attack is illustrated schematically in Figure 2.1.

### **2.2.2 Certificate Pinning**

Android applications, by default, verify the authenticity of the received server certificates when starting a new HTTPS connection. The connection is aborted if this check fails. The check itself compares the received certificate's issuer with the Android list of trusted system Certificate Authorities. However, some applications perform their own certificate validation against a specific custom list of CAs which they have decided to trust. This is known as

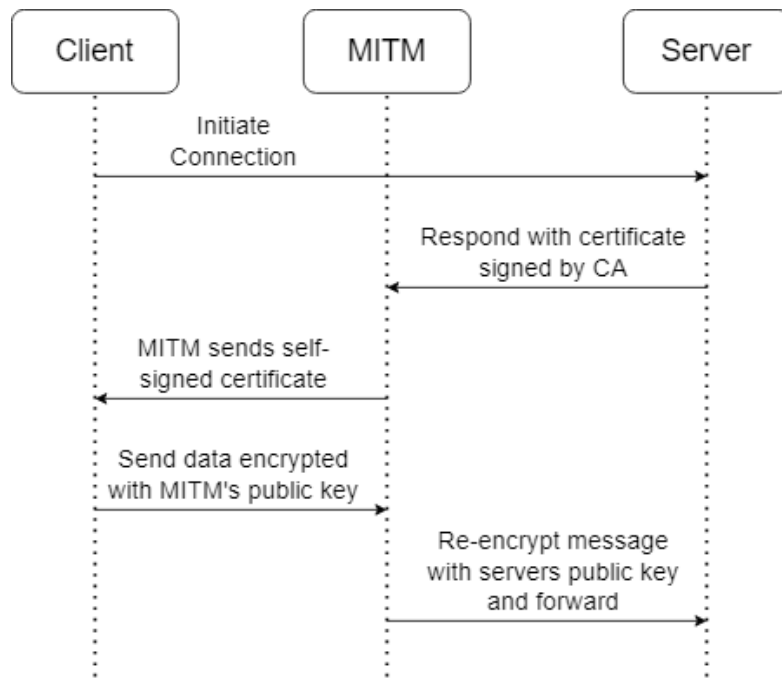


Figure 2.1: The steps carried out in a Man in the Middle attack.

certificate pinning (48), and ensures that no certificate will ever be trusted if it comes from an issuer who is not explicitly trusted by the application.

## 2.3 Data Formats

### 2.3.1 Protocol Buffers

Protocol Buffers, often referred to as Protobufs, are a mechanism for serialising structured data and are developed by Google. They are both forward and backwards compatible and are claimed to be smaller and faster than JSON or XML (23). Protobuf messages are data structures composed of uniquely numbered data fields. The fields themselves consist of key-value pairs, where the key is typically a name, and the value is of a type such as boolean, string etc., or another protobuf message. Protobuf messages are defined within a '.proto' file, and they can be serialised to and from a stream of bytes through methods auto-generated with the proto compiler (12).

The binary version of a message uses the field's number as the key because the name and declared type for each field can only be determined on the decoding end by referencing the message's definition (22). This makes it extremely difficult to interpret the meaning of a protobuf message without having access to the '.proto' definition file.

### 2.3.2 Avro

Avro is a data serialisation framework developed by Apache (7). It relies on a JSON schema for its data definition and stores the data itself in a binary format. The schema and the data are stored together in one file or message. It is not possible to decode an Avro message without access to its schema.

## 2.4 Android

### 2.4.1 Rooting

Rooting an Android phone is the process of obtaining root privileges on the operating system. It is equivalent to jailbreaking on iOS. With root access, it is possible to modify system applications and settings or install software that is otherwise not possible and perform other actions that require administrator permissions. In the context of this work, rooting is necessary in order to install the necessary certificates that are required to perform a man in the middle attack (36).

### 2.4.2 APK

Android Package Kit (APK) is the file format used for distributing and installing Android applications. An APK contains all the required elements for an application to install successfully.

## 2.5 Tools

### 2.5.1 mitmproxy

Mitmproxy is a set of tools that provide an interactive, SSL/TLS-capable intercepting proxy for HTTP/1, HTTP/2, and WebSockets (46). Specifically, the mitmdump command-line tool is used as part of the experimental setup in this work, playing an instrumental role in the man in the middle attack that is performed. It can be thought of as tcpdump for HTTP(S) traffic (46). When using mitmdump in transparent mode, network traffic is redirected to the proxy at the network layer through firewall rules, without any client configuration required. A mobile handset connected to an access point running mitmproxy is therefore unaware of the proxy and trusts its certificates. This allows for all the outgoing connections from the targetted handset to be intercepted, decrypted, logged and forwarded to the actual destination server.

### 2.5.2 jadx

Jadx is an open-source tool that can be used to decompile Dalvik bytecode into Java, provided an APK or zip file (50). It offers many valuable features such as syntax highlighting, a powerful full-text search, and the ability to navigate to the declaration of a function or find references to it within the code.

### 2.5.3 Frida

Frida is a dynamic code instrumentation toolkit (16) that enables hooking onto functions within the code of Android applications and injecting custom JavaScript snippets. In order to use Frida with an Android application, a frida-server needs to be downloaded and run on the targetted handset. Then, to control the device, it needs to be connected via USB to a computer, from which the Frida command-line interface tools can be used to execute commands, such as running a specific mobile application with a particular script that hijacks its functionality.

### 2.5.4 Android Debug Bridge

Android Debug Bridge (adb) is a command-line tool built into Google's Android SDK that enables communication between a client running on a computer and a daemon that runs on a mobile device (3). With adb, it is possible to control the handset from a computer over a USB connection and perform actions such as copying, deleting or altering files, installing APKs or pulling them from the device to the computer. It is also possible to issue shell commands or enter an interactive shell directly.



### 3 Related Work

There has been much work done in the field of user privacy within Android applications, particularly with a significant focus on the privacy threats caused by system permissions. This is because the Android security model for protecting access to sensitive data and system resources is based on the static allocation of permissions to applications. (Applications define the permissions they require inside a manifest file.) As described by Mylonas et al. (47), it is often not clear how an application will use the resources once they have been granted. Within the Android Enterprise Security Paper (1) it is stated that the permission request prompts are designed to give users clarity about the request and provide an opportunity to approve or deny it. However, these prompts often do not clearly describe why they are requesting access to specific user information, thereby making it difficult for users to assess potential risks involved in granting these permissions (17).

Each time that a permission is granted, the user introduces potential vulnerabilities and increases threat likelihood (47). As a result, much of the work in the field of privacy on Android has centred around ways to minimise the risks involved in this process by detecting intruding applications. Gibler et al. proposed AndroidLeaks (17), a static analysis framework that automatically detects potential leaks of sensitive information within Android applications. This is done by creating a mapping between Android API methods within the code and the permissions they require and performing a dataflow analysis to determine if it is possible for information coming from a sensitive source to reach a network sink (a method which requires internet access). Ito et al. (29) developed a method for detecting applications that access sensitive information unrelated to their functionality. This was achieved by modifying the Android source code to record API calls and then performing a statistical analysis of the application's activity logs. Another framework, proposed by Mann and Starostin (41), is based on a similar concept for detecting privacy leaks as Gibler's and also provides a statistical analysis of whether a given application respects its privacy policy. Other works by Barbon et al. (10) and Verderame et al. (54) are also focused on evaluating the compliance of applications with their respective privacy policies.

The work described above makes the significance of system permissions and privacy policies obvious and highlights their relevance to a discussion about user privacy on Android. Thus,

consequently, these factors are also addressed in this dissertation. Despite this, the research of user privacy in Android applications leaves plenty of room for further investigation. This is due to the limitations of static analysis and the lack of user input in work discussed above. Additionally, the work was focused mainly on providing a generic solution for detecting potential privacy leaks without analysing the content of the transmitted data itself. This can likely be attributed to the widespread usage of HTTPS encryption and various other security strategies implemented by applications to secure their data, making this an inaccessible problem that requires an in-depth, hands-on investigation.

Nevertheless, the data sent by applications over the network to their back-end servers has become a target for some recent research and is, in fact, the main focus of this dissertation. Leith and Farrell had laid the foundations for such work with their investigations of the Singapore OpenTrace (Coronavirus contact tracing) application (37), as well as multiple European Google/Apple Exposure Notification (GAEN) based contact tracing applications (38). The experimental setup described in this work relied on a controlled Wi-Fi access point that ran mitmproxy (46), a tool used to enable the man in the middle attack that was performed. The setup can easily be extended for analysis of any other Android applications without requiring much change, and hence it has been adapted for use in this dissertation. While the use of mitmproxy allows for the bypass of HTTPS encryption, it has been found that the data is also often encoded in a binary format which typically lacks public documentation. This is indeed the case with the logs collected for both Gboard and SwiftKey. Therefore, the source code of the corresponding applications needs to be analysed to decode this data. This is not a trivial process because, as described by Graux et al. (27), obfuscating the source code of Android applications is considered standard practice and is done in order to protect applications from malicious users and unwanted exposure. As observed in this dissertation, obfuscation was only one of the security barriers utilised by Gboard and SwiftKey.

Leith's previous work also covers investigations of operating systems themselves, including iOS and Android (34), as well as multiple Android variants, particularly those developed by Samsung, Xiaomi, Huawei, Realme, LineageOS and /e/OS (39). Microsoft's SwiftKey is the default keyboard on Huawei handsets, and while it was not the focus of the report, the keyboard was briefly investigated during the research. It was found that the application encodes its logs in the Avro binary format and also utilises certificate pinning, which was incompatible with the previously described experimental setup. This required the pinning code within the application to be bypassed with the use of custom code injection, which was done with the use of Riru (49) and EdXposed (14). This research laid the groundwork for decoding SwiftKey's logs, and this dissertation follows in its footsteps to provide a deeper dive into the keyboard application. Furthermore, the analysis of the Google Dialer and Messages applications (35) was one of the first to cast light on the telemetry sent by Google Play Services and was instrumental in laying the foundations for a privacy analysis of Gboard. It was found that data is sent to Play Services

via two channels, namely the Clearcut Logger and Firebase Analytics. The Clearcut Logger specifically, sends data to the <https://play.googleapis.com/log/batch> endpoint, containing logs from various Google applications, including Gboard. The log entries present in this data are encoded as a protobuf array. The process used to extract and decode these log entries in Leith's work was adapted and used to investigate logs generated by Gboard.

The privacy of Android keyboards has not been the target of much research. One of the only publications available in this field is from 2015 when Cho et al. took on the study of 139 free third-party keyboards available on the Play Store (13). The work presented was focused on detecting malicious keyloggers. The threats coinciding with the internet permission were also illustrated by showing how a keylogger application could disguise itself as a legitimate keyboard application. Access to the network allows applications to send data to a remote back-end server, and there is often no way for users to find out what data is being sent. Both Gboard and SwiftKey require network permissions and provide no way for users to deny them. The work done by this dissertation investigates the content of the data transmitted by both applications. On the other hand, the third application studied, AnySoftKeyboard, does not require network permissions and thus has no way to transmit any data to a destination outside the device. The work done by Cho et al. is similar to other work described above in that it mainly focuses on the system permissions granted to the studied applications. It provides no investigation into the content of the network traffic generated by these applications. It should be noted that the names of the studied keyboards are not explicitly outlined. The phrasing "third-party keyboards" also likely implies that these applications have lower amounts of users than first-party keyboards, which come pre-installed on specific variants of Android, for example, SwiftKey on Huawei phones or Gboard on Google Pixels. In this dissertation, the choice of keyboard application was based on popularity and amount of installs, as it is believed that studying the most popular applications will have the most significant impact.

## 4 Experimental Setup

### 4.1 Data Collection

The privacy investigation done by this work is focused on the data sent by applications to their back-end servers. The experimental setup described in this chapter is used for capturing the outgoing network traffic from a mobile handset. This data is then manually reverse engineered and analysed, the process of which is further described in detail in Chapter 5.

While it is generally trivial to intercept network packets with tools such as `tcpdump`, this often does not benefit privacy analysis because of the widespread usage of HTTPS encryption. Packet payloads are also commonly encoded in binary formats which lack public documentation. The setup used in this research has previously been successfully used in related work (37) (39). It relies on controlling a Wi-Fi Access Point (AP), which is done on a Raspberry Pi 4 that acts as an AP and runs `mitmproxy` to intercept network requests. Firewall rules on the Pi are set to redirect all HTTP/HTTPS traffic to a port controlled by `mitmproxy`, making the proxy transparent to the target mobile handset. The handset device is connected to the Wi-Fi network created by the Raspberry Pi when running the experiments described below. Any connections going out of the device are intercepted by the `mitmproxy`, which pretends to be the destination server. These connections are logged and subsequently forwarded to their actual destinations. The proxy also follows a similar process for any messages destined from a server to the handset device. This setup is illustrated in Figure 4.1.

In order to bypass the encryption of HTTPS connections, the Android handset was rooted using `Magisk` (53), and a trusted root certificate from the `mitmproxy` Certificate Authority (CA) was installed on it (36). Applications often verify the server certificates' authenticity when starting a new connection and abort this operation if the check fails. When handling a new connection, `mitmproxy` pretends to be the destination server and presents a fake certificate for the target handset. Since a trusted `mitmproxy` root certificate had been pre-installed on the device, the handset accepts the fake certificate and uses the proxy's public key to encrypt the connection data. This allows the proxy to decrypt the network traffic and store it for further analysis. The data is then re-encrypted with the destination server's public key and forwarded accordingly. This process was sufficient to bypass the encryption of Gboard's network traffic;

however, it was not the case with SwiftKey because the keyboard application utilised certificate pinning and would not trust the fake certificate presented by the proxy. Frida was used to overcome this obstacle, the process of which is further discussed in Chapter 5.

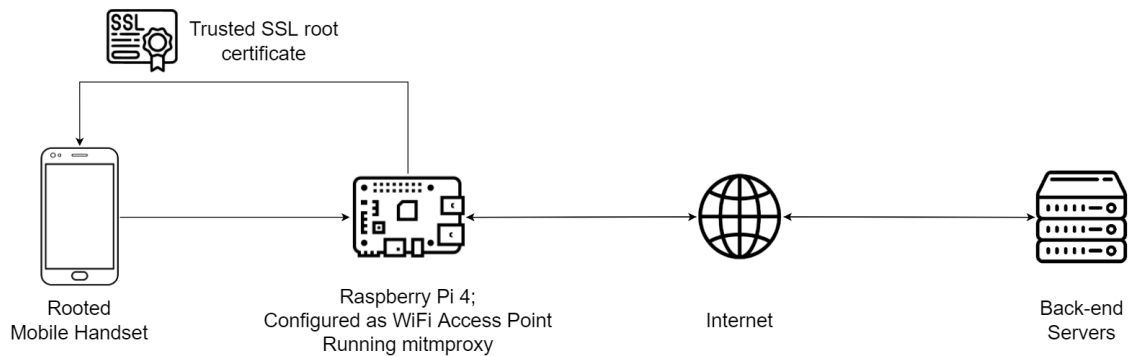


Figure 4.1: The experimental setup used for data collection.

## 4.2 Hardware and Software Used

The experiments performed as part of this research were all done using the same hardware, running the identical versions of relevant software. This allowed for equal comparisons to be made between the keyboards by making the environment as reproducible as possible.

The main handset that was used for conducting each of the experiments can be specified as follows:

- Model: Google Pixel 2
- OS: Android 11 (build RP1A.201005.004.A1)
- Google Play Services version: 16.5.1
- SIM Carrier: None

A second handset was used in one of the experiments to simulate a text conversation between two devices. This handset can be specified as follows:

- Model: Google Pixel 2
- OS: Android 10 (build QP1A.190711.019)
- Google Play Services version: 13.5.1
- SIM Carrier: None

Both devices were rooted with Magisk version 23.0 and ran frida-server version 15.1.14 where required. Neither device had a SIM card inserted during the experiments and did not connect to a mobile network. Each device received a factory reset prior to the experiments. Following

the reset, the device rebooted into a welcome screen and presented various data sharing options and terms and conditions. Only the required terms and conditions were accepted, and the data sharing options were turned off where possible. This setup mimics a privacy-conscious user, who will typically say no if prompted to give consent to data collection. The handsets were also not logged in with a Google account other than to download the relevant applications from the Play Store.

Given that the Android keyboards are live applications that constantly receive updates, it was necessary to stay consistent and always use the same version for each experiment. The following keyboard versions were used in this research:

- Gboard version: 11.1.04.397969183-release-arm64-v8a
- SwiftKey version: 7.8.3.5
- AnySoftKeyboard version: 1.11.7137

A Raspberry Pi 4 Model B Rev 1.2, running Raspbian GNU Linux 11, was used in this research and was configured to be an Access Point by following the official documentation (40). The Raspberry Pi was also running mitmproxy version 7.0.0. The iptables firewall rules were configured to redirect all HTTP(S) traffic to port 8080, as well as reject any UDP traffic on port 443 in order to force any Google QUIC (26) traffic to use TCP, as no tools for decrypting QUIC are available (39).

The primary tools used for reverse engineering were:

- jadx version 1.3.1
- Frida version 15.1.17

## 4.3 Experiments

The data collection process was based on many simple scenarios that could be uniformly applied to all the studied handsets, allowing for direct comparisons. The experiments were designed to be controlled and generate reproducible behaviour. This approach also allows future research to follow the same procedure and gather results for other keyboards, which can be directly compared to this work. The experiments simulate everyday actions that an average user would take when interacting with a keyboard, such as typing text, using voice dictation or gesture typing, and more specific scenarios such as entering a password or navigating the settings screen within the application.

In order to stay consistent, the same four input strings were used in the appropriate experiments. These strings were "The quick brown fox jumps over the lazy dog.", "Lorem ipsum dolor sit amet", "How is the weather today?" and "Hello, World!". These hold no intrinsic

value and were chosen at will. It should be noted that keeping track of the input and the corresponding timestamps proved to be very beneficial during the reverse engineering process, as particular log entries were able to be mapped to specific inputs based on matching timestamps.

The following experiments were carried out for each of the studied keyboard applications:

1. Messaging application (Typing) - WhatsApp

The four input strings were typed in and sent through WhatsApp, one at a time, at 5-minute intervals. Between each text, WhatsApp was closed, and the device was returned to the home screen. The goal of this experiment was to observe what information gets logged when typing on the keyboard. This could then be used as a point of comparison for the remaining experiments.

2. Browser (Typing) - Chrome

The four input strings were typed in and searched for using Chrome, one at a time, at 5-minute intervals. Between each search, Chrome was closed, and the device was returned to the home screen. The goal of this experiment was to detect whether observed results would differ based on the application in which the keyboard was used.

3. Messaging application (Voice) - WhatsApp

The four strings were input through the voice typing feature and sent through WhatsApp, one at a time, at 5-minute intervals. Between each text, WhatsApp was closed, and the device was returned to the home screen. The goal of this experiment was to compare the data generated by voice input to that of gesture input and typing.

4. Messaging application (Gesture) - WhatsApp

The four strings were input through the gesture typing feature and sent through WhatsApp, one at a time, at 5-minute intervals. Between each text, WhatsApp was closed, and the device was returned to the home screen. The goal of this experiment was to compare the data generated by gesture input to that of voice input and typing.

5. Entering Password (Typing) - Pinterest

The Pinterest app was opened, and the login details of an account were entered. Following this, the 'pinterest.ie' website was visited, and the login process was repeated. The goal of this experiment was to determine whether any log entries are generated when entering passwords and if they contain any information which could be linked to the sensitive input.

6. Entering Password (Clipboard Paste) - Pinterest

The same procedure was followed as in Experiment 5, except that the passwords were pasted in from the clipboard rather than being typed in. The goal of this experiment was to find out if pasting in passwords is treated differently than typing them in.

#### 7. Other Language (Typing) - WhatsApp

The same procedure as described in Experiment 1 was followed, except with the use of a French keyboard. The goal of this experiment was to identify what language-specific information, if any, is logged.

#### 8. Messaging application conversation (2 Phones Typing) - WhatsApp

Two phones were used to simulate a text conversation between each other. The devices alternated in sending messages, which mimicked a human interaction of greeting each other and exchanging some information. The goal of this experiment was to determine whether it would be possible to conclude that two devices are communicating based on the logs they generate throughout the conversation.

#### 9. Settings

The settings menu was opened from within the keyboard application, and the available menus were navigated, keeping track of visited pages. The goal of this experiment was to find out if specific application usage, such as visiting a particular screen in the settings, is tracked.

#### 10. Privacy Policy

The privacy policy was opened from the keyboard application's settings. The goal of this experiment was to identify if a user's interaction with the privacy policy would be tracked.

After each experiment, the setup was left running until the desired connections were seen in the mitmproxy logs, as these connections were not always immediate and could take up to a couple of hours to show. It should also be noted that the experiments target the keyboards themselves, and the applications used for entering input have no major significance and were primarily chosen due to their popularity.



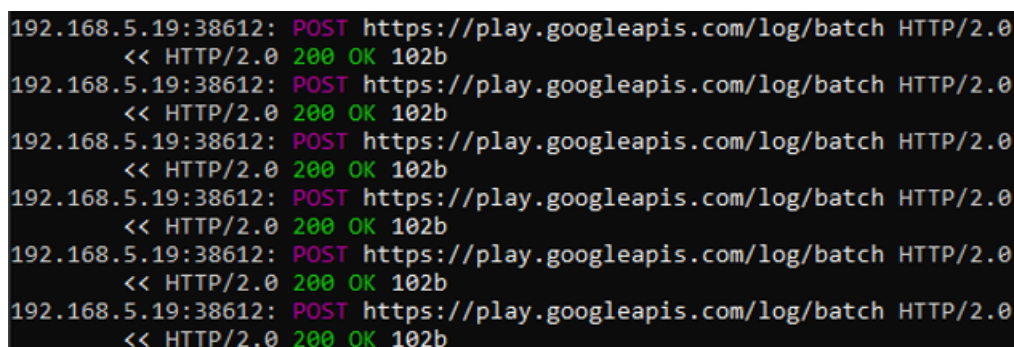
## 5 Reverse Engineering

Reverse engineering is the process of deconstructing and analysing a product or object with the intention of understanding how it has been designed and how it functions. In the context of this work, the Android keyboard applications are the targets of this process, particularly the two proprietary applications, Gboard and SwiftKey. This is because the log traces generated by these applications are encoded in binary formats which are not publicly documented. Therefore, to understand their content, the source code of the logging process needs to be investigated. The difficulty with doing this is that these applications are closed-source. Many tools exist which allow for the decompilation of APKs, such as jadx or Apktool (28), however, the resulting code is nearly always deliberately obfuscated and not human-readable (27). This makes it very difficult to decipher meaning from the code and requires a significant amount of non-trivial reverse engineering work, which must be done manually due to the lack of automated tools. Naturally, applications differ from one to another (and even themselves throughout different releases), especially if they are created by different companies. As such, the reverse engineering work required does not always directly translate from one application to the next, however, underlying principles and techniques can often be carried over. As described in Chapter 3, work previously done on Google’s Messages and Dialer applications (35) laid the foundations for reverse engineering logs sent to Google Play Services, which this work has been able to build on top of when investigating Gboard. Additionally, previous research into the privacy of several Android operating systems (39) briefly investigated SwiftKey since it is the default Huawei keyboard, and this was used as the basis of the reverse engineering efforts in this work.

During the data collection process, it was evident that there was much communication going on between the device and various server endpoints. It is important to note that many of these connections proved to be irrelevant to a discussion about user privacy and, as such, were ignored and omitted from this work. The relevant connections for each studied application and how their corresponding logs were reverse-engineered are addressed in the following sections.

## 5.1 Gboard

The telemetry sent by the Gboard application does not go directly to Google. Instead, the data goes through a service known as the Clearcut Logger within Google Play Services (35). The logger collects the received data and forwards it to Google's back-end servers in batches, specifically the `https://play.googleapis.com/log/batch` endpoint. A mitmproxy console window intercepting and logging the connections made to this endpoint is shown in Figure 5.1.



```
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
192.168.5.19:38612: POST https://play.googleapis.com/log/batch HTTP/2.0
<< HTTP/2.0 200 OK 102b
```

Figure 5.1: Connections to the `play.googleapis.com/log/batch` endpoint.

The content of each connection is structured into several inner batches. These consist of a header and a series of log entries associated with a log source. A sample header is shown in Figure 5.2a. Upon inspection of the content of these batch requests, it was evident that the logs originate from many sources. For example, log source names such as `CLIENT_LOGGING_PROD` or `ANDROID_CHECKIN_METRIC_LOG` were seen, among many others. Gboard itself sends data through multiple log sources, however, the one which has been analysed as part of this research is `LATIN_IME`.

Each log source has a sequence of log entries associated with it. These entries are sent as a binary message that encodes an array of protobuf messages. The array is a sequence of `<length/varint><protobuf>` entries, from which individual protobufs need to be extracted and decoded. The format of individual protobufs in the array depends on the log source.

Protobuf messages can be encoded and decoded using a protobuf definition file. However, no such file is publicly available for Gboard. Instead, the Google Protobuf compiler can be used to decode the message without knowledge of the definition file through the `decode_raw` option (12). The resulting message lacks any interpretation of values and presents ambiguity over the value types - e.g. if a 0 is an integer or a boolean false or an enumerated type. An example of a log entry generated by the `LATIN_IME` source, which was decoded using the protobuf compiler, is shown in Figure 5.2b. Due to the lack of public documentation, the source code of the Gboard application needed to be investigated to make sense of the fields and values present inside the log entries. To do this, the Gboard APK was pulled from the phone using the adb shell and decompiled with jadx. This generated over ten thousand obfuscated Java

```

1  header{
2      always4: 4
3      deviceInfo {
4          androidID: 123456789
5          SDKVersion: 30
6          model: "Pixel 2"
7          product: "walleye"
8          buildID: "RP1A.201005.004.A1"
9          googlePlayServicesVersionString: "297557"
10         hardware: "walleye"
11         device: "walleye"
12         language: "en"
13         country: "US"
14         manufacturer: "Google"
15         brand: "google"
16         board: "walleye"
17         radioVersion: "g8998-00034-2006052136"
18         buildFingerprint: <...>
19         googlePlayServicesVersionCode: 214815037
20         dynamiteModules: <...>
21         accessibilitySettings: <...>
22         buildType: "user"
23         googlePlayServicesVersionName: <...>
24         simCarrierId: -1
25         sdkExtensions: <...>
26     }
27     timestamp: 1642443000
28 }

```

(a) The header of an inner log batch.

```

logEntry:{
  1: 1640538741161
  6 {
    5 {
      1: 180225
      2: 1
      3: 0
      4: "com.whatsapp"
      6: 6
      7: 0
    }
    20: 1
    29 {
      1: 0
      2: 1
      5: "11.1"
    }
    54 {
      1: 0
    }
  }
  11: 9
  15: 0
  17: 329072
  16 {
    2 {
      1: 3
      2: "\0253\327\233\362\000\003\362\004\365"
    }
    3: 1
  }
  22: 14
  23 {
    1: 1
    2: 0
  }
  30: 0x3ff0000000000000
}

```

(b) A sample log entry from the LATIN\_IME source, decoded using Google's Protobuf compiler using the `-decode_raw` option.

Figure 5.2: The content of a log batch connection.

files which needed to be analysed for the protobuf values. Due to the amount of code, it would not be feasible to look through every single file, and a more efficient approach was required. The Clearcut logger was deemed a good starting point for the investigation, and the first step of analysing the code was to find classes relevant to it. Strings within the code do not get obfuscated, which was taken advantage of with jadx's powerful search functionality. Searching for the string "ClearcutLogger" resulted in several matches, one of which was within a class responsible for building log events. The constructor of this class is shown in Figure 5.3 to illustrate the code obfuscation (Note: the obfuscated code will look different on other versions of Gboard). Upon analysis of the log event builder class, it was clear that certain classes were frequently referenced and likely to be of high importance and relevance. However, blindly investigating these classes would be equivalent to navigating a maze in the dark and

would not lead to a deeper understanding of the codebase. In order to combat this, specific objectives needed to be set to keep the investigation focused on the relevant material. It was observed in the log entries that each of them follows the same outer level structure, see fields 1, 6, 11, 15, 17, 16, 22, 23, 30 in Figure 5.2b. This structure will be referred to as the root protobuf, and the first objective was to find where it was defined within the code. After tracing through some of the classes referenced within the log event builder constructor, the class with the root protobuf definition was found. Each protobuf within Gboard has its own definition class and is defined with the compact protobuf format. While there is no public documentation about the compact protobuf format, useful comments are embedded in the Android source code (24), and these comments will be briefly referenced as documentation in the remainder of this section.

```
public gwa(gwe gwe, pox pox, gvy gvy) {
    pqa pqa = (pqa) pzj.j.q();
    this.k = pqa;
    this.a = gwe;
    this.h = gwe.j;
    this.j = gwe.k;
    long currentTimeMillis = System.currentTimeMillis();
    if (pqa.c) {
        pqa.bW();
        pqa.c = false;
    }
    pzj pzj = (pzj) pqa.b;
    pzj.a = 1 | pzj.a;
    pzj.b = currentTimeMillis;
    long d = gla.d(((pzj) pqa.b).b);
    if (pqa.c) {
        pqa.bW();
        pqa.c = false;
    }
    pzj pzj2 = (pzj) pqa.b;
    pzj2.a |= 65536;
    pzj2.g = d;
    if (ibn.b(gwe.e)) {
        boolean b = ibn.b(gwe.e);
        if (pqa.c) {
            pqa.bW();
            pqa.c = false;
        }
        pzj pzj3 = (pzj) pqa.b;
        pzj3.a |= 8388608;
        pzj3.h = b;
    }
    if (SystemClock.elapsedRealtime() != 0) {
        long elapsedRealtime = SystemClock.elapsedRealtime();
        if (pqa.c) {
            pqa.bW();
            pqa.c = false;
        }
        pzj pzj4 = (pzj) pqa.b;
        pzj4.a |= 2;
        pzj4.c = elapsedRealtime;
    }
    if (pox != null) {
        if (pqa.c) {
            pqa.bW();
            pqa.c = false;
        }
        pzj pzj5 = (pzj) pqa.b;
        pzj5.a |= 1024;
        pzj5.f = pox;
    }
    this.b = gvy;
}
```

Figure 5.3: Obfuscated constructor of a Clearcut Logger class responsible for building log events.

A compact protobuf definition relies on a String object and an Object[] array of variables. The String object encodes a sequence of integers into UTF-16 characters. The first ten of these characters are metadata, and the remainder of the object contains field entries outlining the numbers and types of fields from the protobuf. The Object[] array contains variables from the definition class, ordered in the same way as they appear in the String object and are mapped accordingly. To explain by example, the compact protobuf definition String object and Object[] array found within the log event builder class are shown below.

```
"\u0001\u0007\u0000\u0001\u0001\u001c\u0007\u0000\u0000\u0000
\u0001\u1002\u0000\u0006\u100a\n\u000b\u1004\u0004\u000f\u1010
\u0010\u0011\u1002\u0001\u0019\u1007\u0017\u001c\u1008\u0018",
new Object[]{"a", "b", "f", "e","g", "c", "h", "i"});
```

The first ten UTF-16 characters represent values such as how many fields are expected in the Object[] array, the minimum and maximum numbers of those fields etc. Then, the characters are grouped in threes, following the structure <field number><field type><hasbits offset>. The documentation contains a description of the meaning of each value in the String object and a mapping for field types, which are represented by integers and are mapped to over sixty different value types. Table 5.1 shows the field entries within the compact protobuf String object mentioned above (i.e. excluding the first ten UTF-16 characters). Each entry is then translated into its field number and field type and shown which variable it is mapped to. For example, the first entry \u0001\u1002\u0000 is translated in the following way: \u0001 -> field number 1, \u1002 -> translates to decimal 2, which according to the documentation maps to field type int64. The third value \u0000 can mostly be ignored here and is unrelated to the variable mapping. This field entry maps on to variable 'b' because 'b' is the first variable in the Object[] array (the first variable 'a' is ignored by default).

	Field number	Field type	Variable
\u0001\u1002\u0000	1	int64	b
\u0006\u100a\n	6	bytes	f
\u000b\u1004\u0004	11	uint32	e
\u000f\u1010\u0010	15	sint64	g
\u0011\u1002\u0001	17	int64	c
\u0019\u1007\u0017	25	boolean	h
\u001c\u1008\u0018	28	string	i

Table 5.1: The field entries found within the compact protobuf definition of the log entry root protobuf.

Translating the compact protobuf definition reveals which fields from the log entries map to which variables inside the code. Despite removing the ambiguity over the value type, it does not reveal any information as to what the value itself might represent. To figure out the meaning of a particular field, the corresponding variable must be traced through the code to

the point where it is either initialised or assigned a value. This can be done with jadx's helpful "Find usage" tool, which lists all references to a variable or function found within the code. It can then be seen what values are assigned to the variable, and meaning can be deduced from this. This can sometimes be a straightforward process, as the variable might be assigned an obvious value such as "System.currentTimeMillis()". Other times this is not so trivial as the value might, for example, be an integer that is mapped onto an enumerated type, which would then have to be tracked down itself. This work has to be done manually for each variable that contributed to the log entries due to the lack of tools to automate the process.

It is also often the case that variables contain objects of another protobuf definition class, meaning the message contains nested protobufs. An example of this is field 6 of the above-defined protobuf, which can be seen in Figure 5.2b to contain multiple nested protobuf messages. It turns out that this field's definition contains over eighty nested protobuf messages. However, many of these were not found within the logs that were collected; therefore, these protobufs were excluded from the investigation. Due to the sheer size of the codebase and the limitations of manual investigation, it would be impossible to analyse each protobuf within the available time. Additionally, protobuf definitions are also backwards compatible. Some of the fields that appear in the logs may be redundant in newer versions of Gboard but are kept for compatibility with legacy systems. These typically hold the appropriate default value for their type, e.g. 0 for integers. Due to these reasons, this research does not provide an exhaustive reverse engineering of the whole Gboard application. Instead, protobuf messages which appeared consistently within the observed logs were investigated. The focus was mainly on fields that would change based on different user input, as these were deemed more relevant to a discussion about user privacy than constant values. As mentioned earlier, each log source generates a sequence of log entries. These are all shaped by the root protobuf but contain different nested messages. Two of the most interesting log entries observed are shown in Figure 5.4a, with a side-by-side comparison of their decoded values. It should be noted that the names of the fields used have been adapted from their code variable names for clarity and readability. The actual values of each field are taken directly from the source code. For example on line 12 of Figure 5.4b, the value "IMS\_ON\_START\_INPUT\_VIEW" can be seen. This is one of the values from the keyboard events enumerated type used by Google. A more comprehensive (but likely not complete) list of keyboard events that were observed is shown below in Table 5.2.

A substantial number of protobufs were investigated and decoded by finding and interpreting their compact protobuf definition and then tracing the usage of relevant variables within the code to understand the meaning behind the values. As can be seen in the decoded log entries in Figure 5.4b, the root protobuf consists of some timing information, several nested messages and a so-called keyboard event. The value of this field is a number, which is mapped to an activity such as opening the keyboard, typing text, or selecting one of the text suggestions

```

1  ✓ logEntry:{
2    1: 1640538741161
3  ✓ 6 {
4  ✓ 5 {
5    1: 180225
6    <...>
7    4: "com.whatsapp"
8    <...>
9    }
10   <...>
11  }
12  11: 9
13  15: 0
14  17: 329072
15  <Metadata added by Play Services>
16  }
17
18  ✓ logEntry:{
19    1: 1640539630344
20  ✓ 6 {
21  ✓ 2 {
22    1: 5
23    2: 0
24    <...>
25    9: 0
26    11: 0
27    15: 5
28    24: 1
29    <...>
30    31: 1431
31    32: 0xab9c138c8de4086e
32    <Metadata added by Play Services>
33  }
34  ✓ 24 {
35    1: "en"
36  }
37  <...>
38  }
39  11: 3
40  15: 0
41  17: 1218255
42  <Metadata added by Play Services>
43  }

```

(a) The encoded content of a log entry.

```

1  logEntry:{
2    CurrentTimeMillis: 1640538741161
3    NestedLogs {
4      KeyboardUsageInfo {
5        InputType: TEXT
6        <...>
7        ApplicationName: "com.whatsapp"
8        <...>
9      }
10     <...>
11   }
12   KeyboardEvent: IMS_ON_START_INPUT_VIEW
13   TimeZoneOffset: 0
14   ElapsedRealtime: 329072
15   <Metadata added by Play Services>
16 }
17
18 logEntry:{
19   CurrentTimeMillis: 1640539630344
20   NestedLogs {
21     InputWordInfo {
22       WordLength: 5           // Hello
23       GestureInput: False
24       <...>
25       DeletedComposingText: False
26       VoiceInput: False
27       CommittedTextLength: 5
28       TypeOfInput: TEXT
29       <...>
30       TimeToEnterWordInMs: 1431
31       RandomThreadId: 0xab9c138c8de4086e
32       <Metadata added by Play Services>
33     }
34     LanguageInfo {
35       KeyboardLanguage: "en"
36     }
37     <...>
38   }
39   KeyboardEvent: COMMIT_TEXT
40   TimeZoneOffset: 0
41   ElapsedRealtime: 1218255
42   <Metadata added by Play Services>
43 }

```

(b) The decoded content of a log entry.

Figure 5.4: Side-by-side comparison of an encoded log entry and its decoded meaning.

Keyboard Event	Event Number
IMS_CREATED_BEFORE_USER_UNLOCKED	1
SHARED_PREFERENCE_CHANGED	2
COMMIT_TEXT	3
IME_TEXT_CANDIDATE_SELECTED	4
IMS_INPUT_VIEW_CREATED	8
IMS_ON_START_INPUT_VIEW	9
IMS_INPUT_VIEW_FINISHED	10
TEXT_AUTOCORRECTED	11
PERIODIC_TASK_SERVICE_RUN	13
INVALID_WORD_UNDERLINED	14
INPUT_METHOD_ENTRY_CHANGED	16
SETTINGS_ACTIVITY_CREATED	18
SYNC_STATS_RECORDED	19
NEXT_WORD_SUGGESTION_SELECTED	22
DELETED_WORD	25
EDIT_WORD	26
TEXT_CANDIDATES_APPENDED	41
VOICE_INPUT_START	42
VOICE_INPUT_STOP	43
SUPERPACKS	50
ACCESS_POINT_FEATURE_CLICKED	54
OPEN_ACCESS_POINTS	56
PRESSED_SPACE	103
REMOVED_SPACE	104
KEYBOARD_MODE_CHANGED	110
IMS_INPUT_VIEW_STARTED	111
KEYBOARD_ACTIVATED	130
RATEUS_USAGE	148
KEYBOARD_BODY_SHOWN	167
KEYBOARD_SHOWN_LATENCY	168
LANGUAGE_MODEL_LOAD_INFO	197
ON_START_RECOGNITION	215
ON_STOP_RECOGNITION	216
TAB_OPEN	223
IMPRESSION	230
ABANDON_CLIENT_DUE_TO_STALE_CLIENT_REQUEST	238
SUGGESTION_DISPLAYED	242
SUGGESTION_CLICKED	243
PINNED_ACTION_DISPLAYED	244
PINNED_ACTION_CLICKED	245
EMOJI_PREDICTOR_MODEL_LOADED	246
RUNTIME_PARAMETERS_IME_ONACTIVATE	264
SUGGESTION_LONG_PRESSED	274

Table 5.2: The Keyboard Events present in the log entries from the LATIN\_IME source.



prompted by Gboard. Some of the most significant fields which were found within the nested protobufs include the name of the application where Gboard was used, the length of individual words that were typed and how long it took the user to enter them, along with the type of input and language(s) used. A protobuf definition file has been recreated based on the reverse engineering work that has been done and can be found in (31). Chapter 6 outlines the results obtained from performing the experiments and provides commentary about the choices that users have to opt out of this data collection.

## 5.2 SwiftKey

The reverse engineering efforts required to decode SwiftKey logs were drastically different from those required by Gboard. Not only did SwiftKey encode its logs in a different binary format, but it also proposed new security obstacles, particularly certificate pinning. To begin with, when the SwiftKey application was launched during the data collection phase, it was immediately apparent that it was communicating heavily with back-end servers. Connections to endpoints such as `in.appcenter.ms/logs` and `telemetry.api.swiftkey.com` were observed, as illustrated in Figure 5.5. The first of these endpoints is used for sending crash analytics, and the second, as implied by the name, is used for sending telemetry data (39). It should be noted that although SwiftKey asks the user for permission to collect telemetry when setting up the application after a fresh install, these connections appear regardless, even if the user says no. This means that there is no way to opt out of this data collection. If the telemetry collection was agreed to, connections to another endpoint, `snippetdata.api.swiftkey.com/v1/sk-snippetdata`, also began to appear. The focus of this research, and hence the reverse engineering efforts, was around decoding the `telemetry.api.swiftkey.com` connections as they were deemed the most intrusive to user privacy, considering the fact that no opt-out options are available.

```
192.168.5.100:40283: POST https://in.appcenter.ms/logs?api-version=1.0.0
                  << 200 OK 138b
192.168.5.100:43764: clientconnect
192.168.5.100:43764: POST https://telemetry.api.swiftkey.com/v1/bark-logs
                  << 204 No Content 0b
192.168.5.100:43764: POST https://telemetry.api.swiftkey.com/v1/bark-logs
                  << 204 No Content 0b
192.168.5.100:38922: clientdisconnect
192.168.5.100:43764: POST https://telemetry.api.swiftkey.com/v1/bark-logs
                  << 500 Internal Server Error 31b
192.168.5.100:43764: clientdisconnect
192.168.5.100:41827: clientconnect
192.168.5.100:43511: GET https://connectivitycheck.gstatic.com/generate_204
                  << 204 No Content 0b
192.168.5.100:41827: POST https://telemetry.api.swiftkey.com/v1/bark-logs
                  << 204 No Content 0b
```

Figure 5.5: Connections observed on startup of the SwiftKey application.

The first challenge to overcome was that the SwiftKey connections were getting rejected when using the standard experimental setup because SwiftKey did not trust the mitmproxy's fake certificate. This is because the keyboard application performs what is known as certificate pinning, where it restricts which certificates are considered valid and only entrusts certificates provided by a custom list of approved Certificate Authorities. To bypass this, SwiftKey's APK was pulled from the handset using adb and decompiled with jadx. The resulting source code was obfuscated and required manual investigation. The objective was to find where the certificate pinning occurs within the code and bypass it by injecting a custom script. It was found that SwiftKey had compiled okhttp3 (51) into its source code, meaning the library code also got obfuscated. However, the source code for okhttp3 is publicly available online and could be mapped to the obfuscated version by comparing code structure and internal strings. Microsoft was also doing some custom certificate checks on top of the work done by okhttp3. These were found by searching for strings such as "CertificateExpiredException" or "pins matched" within the code. Imitating the success of these checks, e.g. always return true if the method returns a boolean, proved to be enough to bypass the certificate pinning done by the application. This code was dynamically injected into SwiftKey through the use of Frida.

Introducing Frida into the experimental setup required the mobile handset to be connected to a computer by USB. Then, frida-server was installed and ran on the device through the adb shell. Subsequently, on the computer to which the handset was connected, Frida was run with a command that launched the SwiftKey application on the mobile handset and ran the custom script to bypass certificate pinning. The script would dynamically hook onto the targetted functions if the application called them, and substitute code was run instead of the SwiftKey code. This successfully bypassed SwiftKey's certificate pinning functionality and allowed the telemetry connections to be made. The setup described above is illustrated schematically in Figure 5.6. The next challenge was to decode the content of the telemetry connections, which are encoded using the Apache Avro serialisation format (39). Avro requires a JSON schema to be decoded, and unlike protobufs, it is not possible to decode it without the schema. To extract the schema from SwiftKey, Frida was used to dynamically hook on the 'onStart' activity event (2), and execute a 'getSchema()' call (8). With the certificate pinning bypassed, and the schema obtained, the content of the telemetry connections could now be decoded. This could be done for the whole payload at once and, unlike in Gboard, did not require manual investigation of individual values or variables.

The logs within the telemetry connections are a sequence of JSON objects which follow a structure defined within the Avro schema. Each log entry contains similar information at the beginning, namely the installation ID, application version, a timestamp and an offset. After this, the logs begin to differ, with some focusing on application settings and others on the hardware information of the user's handset or the user's interactions with the keyboard. The

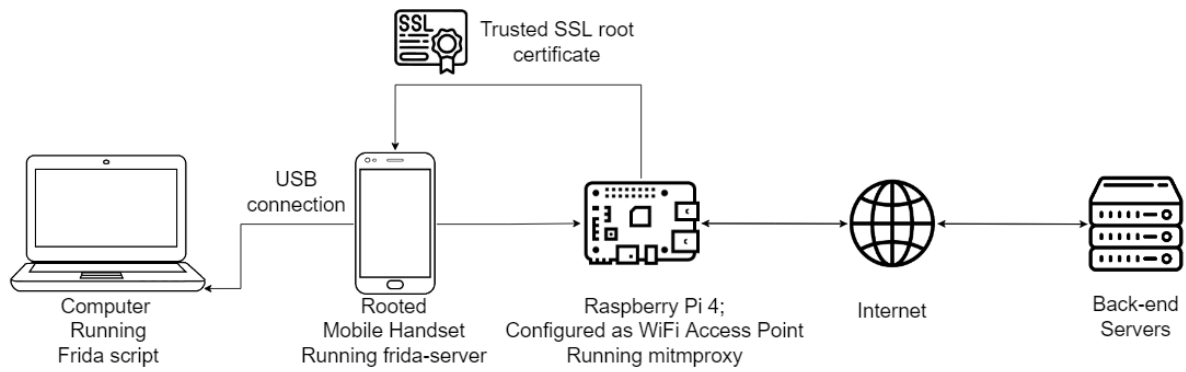


Figure 5.6: The experimental setup used to bypass SwiftKey's certificate pinning.

data collection observed in these logs appears to be quite excessive, especially considering that users have no option to opt out. In summary, some of the most significant things that can be seen in the logs are: handset hardware information, Google advertising ID, a large number of keyboard events, application name where SwiftKey was used, timestamped user typing statistics, including the number of characters entered and the typing duration, the languages that were used, and how many terms per language were entered, among others. Figure 5.7 below shows a summarised version of the log entries observed (Note: these logs were generated by SwiftKey but have been modified to be more presentable). An outline of the results obtained in each experiment, as well as an overall analysis of SwiftKey's privacy, is presented in Chapter 7.

```

1  {'metadata': {'installId': <...>, 'appVersion': '7.8.3.5', 'timestamp': <...>}}
2  √ { <...> }, 'productInfo': {'product': 'SWIFTKEY_ANDROID', 'productId':
3  | | | 'com.touchtype.swiftkey', 'productVersion': '7.8.3.5'}}
4
5  √ { <...> }, 'deviceInfo': {'os': {'name': 'ANDROID', 'version': '11'},
6  | | | 'model': 'Pixel 2', 'manufacturer': 'Google', 'architecture': 'arm64-v8a',
7  | | | 'cpus': 8, 'totalRam': <...>,
8  | | | 'screenMetrics': {'density': 420, 'width': 1080, 'height': 1794},
9  | | | 'deviceId': '', 'operator': {'name': '', 'country': 'ie'},
10 | | | 'locale': 'en_IE', 'language': 'en', 'advertisingId': '123456789', <...>},
11 | | | 'campaign': 'google-play', 'creative': None, 'cohort': None}}
12
13 √ { <...> }, 'name': 'TELEMETRY_PERIODIC_SEND', 'result': 'SUCCESS',
14 | | | 'durationMs': 3584}}
15 { <...> }, 'durationMs': 0, 'inputLength': 10, 'sampleRate': 0.009999999776482582}}
16 { <...> }, 'durationMs': 1, 'sequenceLength': 1, 'sampleRate': 1.0}}
17 √ { <...> }, 'layout': 'qwerty', 'keyboardMode': 'FULL', 'dockState': 'DOCKED',
18 | | | 'layoutTriggerSource': 'START_INPUT_VIEW'}}
19 √ { <...> }, 'orientation': 'PORTRAIT', 'hasFullAccess': True,
20 | | | 'isHardKeyboardConnected': False}}
21 { <...> }, 'trigger': 'AUTOMATIC'}}
22
23 √ { <...> }, 'application': 'com.whatsapp', 'durationMillis': 26004,
24 | | | 'typingStats': {'totalTokensEntered': 5, 'tokensFlowed': 0,
25 | | | 'tokensPredicted': 0, 'tokensCorrected': 0, 'tokensVerbatim': 5,
26 | | | 'tokensPartial': 0, 'netCharsEntered': 28, 'deletions': 0,
27 | | | 'characterKeystrokes': 0, 'predictionKeystrokes': 0, 'emojisEntered': 0,
28 | | | 'remainderKeystrokes': 0, 'predictionSumLength': 0,
29 | | | 'typingDurationMillis': 4157, 'totalTokensEnteredEdited': 0,
30 | | | 'tokensFlowedEdited': 0, 'tokensPredictedEdited': 0,
31 | | | 'tokensCorrectedEdited': 0, 'tokensVerbatimEdited': 0,
32 | | | 'tokensPartialEdited': 0}, 'languagesUsed': 1, 'termsPerLanguage':
33 | | | {'en_GB': 5}, 'tokensPerSource': {'': 3, 'en_GB/en_GB.lm': 1,
34 | | | 'en_GB/en_GB_word_c.lm1': 1}, 'tokensShownPerSource': {...}}

```

Figure 5.7: A summary of the decoded logs sent by SwiftKey to the telemetry endpoint.

## 6 Gboard

### 6.1 Application Overview

Gboard is Google's keyboard application, and it comes pre-installed on many Android devices. It totals over five billion installs on the Google Play Store. The keyboard offers gesture and voice typing, support for over five hundred languages, and Google features such as translate or search, among many others. It is one of the most popular keyboard applications available on Android, yet little is known about its privacy implications.

### 6.2 Permissions

There are four permissions that users can control inside the app settings for Gboard. These are the Camera, Contacts, Microphone and Voice, all of which are denied by default. Allowing specific permissions unlocks more of the keyboard's functionality, e.g. permitting microphone access allows for voice typing. However, it is unclear how these permissions are used once granted and how they handle user data. Descriptions of the permissions can be found within the application settings, and on the Gboard Play Store page (19), but these are presented in a way that is not clear to the users. For example, the Microphone permission has a description stating that the Gboard application can "record audio", with the detailed description stating that "This app can record audio using the microphone at any time". It is not explained why this might happen or what happens with the recording. This could be off-putting to users, who are left in the dark regarding how their data is handled.

The complete list of all permissions requested by the Gboard application is presented in Table 6.1. Several potential privacy concerns are present, most notably the requirement of total internet access, which cannot be denied. This allows the application to communicate with and send data to remote servers, which is, in fact, observed to take place. Furthermore, questions should be raised about why it is necessary for a keyboard application to be able to download files without notifying the user, as well as access their camera and record audio.

Android Permission
ACCESS_NETWORK_STATE
ACCESS_WIFI_STATE
DOWNLOAD_WITHOUT_NOTIFICATION
GET_ACCOUNTS
INTERNET
READ_CONTACTS
READ_PROFILE
READ_USER_DICTIONARY
RECEIVE_BOOT_COMPLETED
USE_CREDENTIALS
VIBRATE
WRITE_USER_DICTIONARY
GET_PACKAGE_SIZE
WAKE_LOCK
com.google.android.providers.gsf.permission.READ_GSERVICES
READ_EXTERNAL_STORAGE
RECORD_AUDIO
FOREGROUND_SERVICE
CAMERA
com.bitstrips.emoji.provider.READ
INTERACT_ACROSS_PROFILES

Table 6.1: The list of permissions requested by Gboard.

## 6.3 Privacy Policy

It is possible to access a privacy policy within the Gboard application's settings under the "About" screen. By default, this policy is opened in Google Chrome, requiring the user to accept Chrome's terms of service to proceed. The policy itself (21) is generic and applies to all of Google's services, not Gboard specifically. Due to its nature, it is not made clear to users which applications or services collect what data. Google does explain, though, that regardless of whether a user is logged into a Google account or not, the collected data will be linked to them. The examples which Google presents are purposefully vague and make frequent use of the terms "may collect" rather than "will" or "do". Therefore, there is no way that users can gain any clarity about Gboard's data collection based on reading the privacy policy provided within the application itself.

Interestingly, the store page for Gboard available on Apple's App Store (18) includes a description with information about "What Gboard sends to Google", whereas this is not found on Google's Play Store page (19) available on Android. This description states that searches to Google's web servers, usage statistics, and voice inputs are sent to Google. Similarly to Gboard's advanced settings, it is not made clear what is included as part of the usage statis-

tics. It is also claimed that Gboard does not send anything else about what is typed to Google and that the dictionary data built up and kept by Gboard is stored locally and never leaves the device. While not completely transparent, this information helps users be more aware of how their data is handled while using Gboard and would be a lot more valuable if it was available on all platforms.

## 6.4 Opting Out

It is possible to opt out of the data collection done by Gboard. This can be done inside the application's advanced settings, shown in Figure 6.1, by turning off the "Share usage statistics" option. It is worth noting here that this is an opt-out, not an opt-in, meaning that the setting is turned on by default, rather than asking the user upfront about their preference. The setting description is very ambiguous and unclear. There is also no option available to "Learn more", as can be seen under other options such as "Improve voice and typing for everyone". There is no definition for what "usage statistics" might be or what they are used for, leaving users confused about what this setting really does. This does not allow a user to make an informed decision about their consent to this collection of data. Opting out of this setting reduced the LATIN\_IME logs only to contain a single log entry, which consisted of a timestamp, the keyboard event "IMS\_INPUT\_VIEW\_STARTED", the Gboard version and keyboard layout, as well as a couple of other fields which were deemed insignificant.

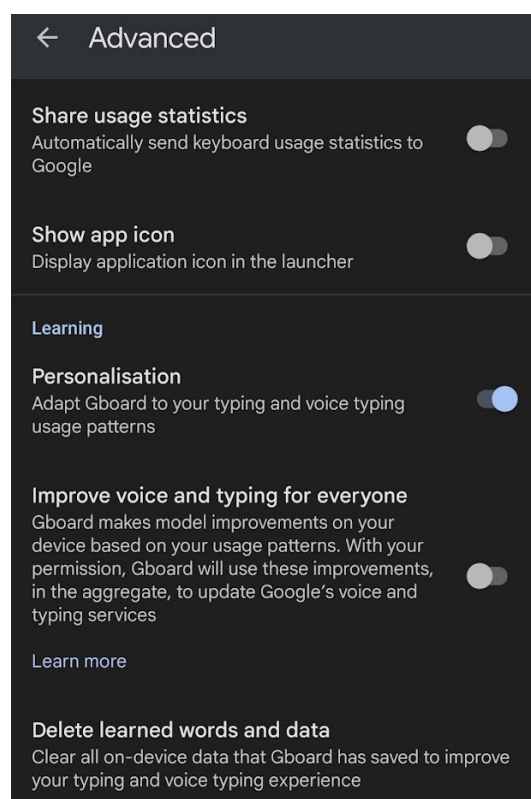


Figure 6.1: The advanced settings available on Gboard.

## 6.5 Experiment Results

The experiments outlined in Chapter 4 were performed on the Gboard application, with the "Share usage statistics" setting turned on. There were some similarities observed across all the collected log entries - these are first outlined, followed by the individual results of each experiment.

Firstly, each inner log batch present in the payload of the `play.googleapis.com/log/batch` connections contains a header. These headers mainly include information about the hardware of the handset, as well as the version of Google Play Services, SIM carrier ID, and most notably, the Android ID, which has been shown to be uniquely identifiable (34).

```
header{
  deviceInfo {
    androidID: 123456789
    SDKVersion: 30
    model: "Pixel 2"
    product: "walleye"
    buildID: "RP1A.201005.004.A1"
    googlePlayServicesVersionString: "67384622"
    hardware: "walleye"
    device: "walleye"
    language: "en"
    country: "US"
    manufacturer: "Google"
    brand: "google"
    board: "walleye"
    buildFingerprint: "google/walleye/..."
    <...>
    buildType: "user"
    googlePlayServicesVersionName {
      verMajor: 16
      verMinor: 5
      verMinor2: 1
    }
    simCarrierId: -1
  }
  timestamp: 1640527800
}
```



Whenever the keyboard is opened inside an application, a log entry is generated, containing its name and editor input type. This information is nested within the `input_info` field of the root protobuf. The remaining fields, `currentTimeMillis`, `keyboardEvent`, `timeZoneOffset` and `elapsedRealtime`, are also part of the root protobuf and appear in all log entries.

```
logEntry:{
  currentTimeMillis: 1640538741161
  input_info {
    keyboard_usage_info {
      editorInfoInputType: TEXT
      <...>
      applicationName: "com.whatsapp"
      <...>
    }
    <...>
  }
  keyboardEvent: IMS_ON_START_INPUT_VIEW
  timeZoneOffset: 0
  elapsedRealtime: 329072
  <Metadata added by Play Services>
}
```

#### 1. Messaging application (Typing) - WhatsApp

It was observed that a timestamped log entry was created for each word entered. Inside these log entries, the length of the word, the type of input, the time it took to enter the word and a random identifier were present. The value of this identifier could not be linked to any of the other log entries, even those corresponding to the same word being entered, suggesting that it is unique. The keyboard event which corresponds to these log entries is `"COMMIT_TEXT"`. The languages used by the keyboard were also observed to be logged. No evidence was found to suggest that Gboard tracks which characters have been entered or their frequency.

#### 2. Browser (Typing) - Chrome

The results obtained were similar to those described in Experiment 1. It appears that the generated logs are not affected by the application in which Gboard is used.

```

logEntry:{
  currentTimeMillis: 1640539630344
  input_info {
    word_input_info {
      characterSequenceLength: 5 // Hello
      isGestureInput: False
      <...>
      TEXT_COMPOSING_DELETED: False
      isVoiceInput: False
      committedTextLength: 5
      typeOfInput: TEXT
      <...>
      timeToEnterWord: 1431
      randomThread: 0xab9c138c8de4086e
      <Metadata added by Play Services>
    }
    language_info {
      keyboardLanguage: "en"
    }
    <...>
  }
  keyboardEvent: COMMIT_TEXT
  timeZoneOffset: 0
  elapsedRealtime: 1218255
  <Metadata added by Play Services>
}

```

### 3. Messaging application (Voice) - WhatsApp

Similar to the first experiment, the "word\_input\_info" log entries were present, and they specified the input type of the messages to be voice. Keyboard events representing "VOICE\_INPUT\_START" and "VOICE\_INPUT\_STOP" were also observed, both of which correspond to a timestamp, meaning it is possible to interpret how long the voice typing function was used for. However, the fields corresponding to the time taken to enter an individual word and the languages used were not observed in the results of this experiment.

```

logEntry:{
  currentTimeMillis: 1640618474657
  <...>
  keyboardEvent: VOICE_INPUT_START
  <...>
}

logEntry:{
  currentTimeMillis: 1640618483942
  input_info {
    word_input_info {
      characterSequenceLength: 5 // Hello
      isGestureInput: 0
      TEXT_COMPOSING_DELETED: 0
      isVoiceInput: 1
      committedTextLength: 5
      typeOfInput: VOICE
      <...>
      randomThread: 0x42fcf4f473f74886
      <Metadata added by Play Services>
    }
    <...>
  }
  keyboardEvent: COMMIT_TEXT
  timeZoneOffset: 0
  elapsedRealtime: 546061
  <Metadata added by Play Services>
}

logEntry:{
  currentTimeMillis: 1640618488224
  <...>
  keyboardEvent: VOICE_INPUT_STOP
  <...>
}

```

#### 4. Messaging application (Gesture) - WhatsApp

Results largely similar to Experiment 1 were observed, with the relevant fields showcasing that the input was through gesture typing.

#### 5. Entering Password (Typing) - Pinterest

Based on the observed results, it appears that password length is not tracked when logging into applications or websites inside the browser. A "word\_input\_info" entry was generated containing the length of the email used to log in, but no entry corresponding to the password was seen.

#### 6. Entering Password (Clipboard Paste) - Pinterest

Typing in the email and pasting the password from the clipboard was observed to generate the same logs as described in the above experiment.

#### 7. Other Language (Typing) - WhatsApp

When using different languages, the logs are updated accordingly. Based on the results of this experiment, it can be concluded that it is the language of the keyboard used, not the word entered, that is present in the telemetry.

```
logEntry:{
  currentTimeMillis: 1640539630344
  input_info {
    <...>
    language_info {
      keyboardLanguage: "fr"
    }
    <...>
  }
  <...>
}
```

#### 8. Messaging application conversation (2 Phones Typing) - WhatsApp

This experiment was conducted to investigate whether it would be possible to link two handsets to be communicating with each other based on the logs generated by their keyboards. Both handsets generated logs similar to those described in Experiment 1 during a simulated conversation. Based on this, it can be concluded that it would be theoretically possible to link the handsets based on the timing information of when messages are sent. This concept is further expanded on in Chapter 9.

#### 9. Settings

The activity of opening the Gboard settings gets logged with a corresponding keyboard event. The event "SETTINGS\_ACTIVITY\_FINISHED" was also found within the source code but was not observed in the collected data. Additionally, no evidence was found which would suggest that Gboard keeps track of which settings screens have been visited.

```
logEntry:{
  currentTimeMillis: 1642357812616
  <...>
  keyboardEvent: SETTINGS_ACTIVITY_CREATED
  timeZoneOffset: 0
  elapsedRealtime: 77871
  <Metadata added by Play Services>
}
```

#### 10. Privacy Policy

The Gboard privacy policy can be accessed through Gboard's settings. However, only the activity of opening the settings is logged, as shown above. The act of accessing the privacy policy itself does not appear to be tracked.

## 7 SwiftKey

### 7.1 Application Overview

SwiftKey is Microsoft's offering to the Android keyboard market, and it totals over one billion installations on the Play Store (43). It is also the default keyboard on Huawei's handsets. SwiftKey is widely considered as the superior keyboard in terms of its word predictions and suggestions. It is known to learn fast and can be synced across multiple devices by logging in to an account. However, as with Gboard, not much is understood about what data is collected and shared by the keyboard.

### 7.2 Permissions

Within the applications settings, users can control three permissions, namely Contacts, Microphone and Storage, which are all denied by default. Similarly to Gboard, there is an issue of transparency present, where the intentions behind the permissions are not made clear to the users. The complete list of requested permissions by the application is presented in Table 7.1. The permissions requested appear to be relatively reasonable for a keyboard application. The main concerns present are yet again full network access, which cannot be denied, and the ability to record audio.

### 7.3 Privacy Policy

The privacy policy can be accessed from SwiftKey's privacy settings, as seen in Figure 7.1b. Similarly to Google, Microsoft offers a generic privacy statement rather than one specific to the keyboard application. In the statement (42), Microsoft outlines that it collects data about users' interactions with their products and provides a bullet point list of potential pieces of information which may be collected. This makes it easy to read, although not all the descriptions are clear. For example, "typing data" is not expanded on and is left to the reader's interpretation. It is also claimed that it is possible to decline when asked to share personal data.

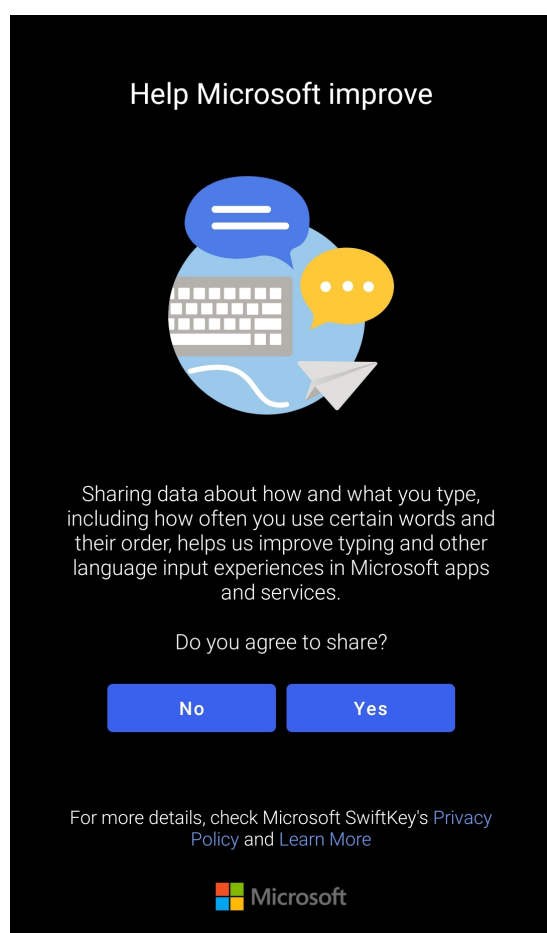
Android Permission
WAKE_LOCK
VIBRATE
WRITE_EXTERNAL_STORAGE
INTERNET
ACCESS_NETWORK_STATE
ACCESS_WIFI_STATE
RECEIVE_BOOT_COMPLETED
GET_ACCOUNTS
com.swiftkey.languageprovider.READLANG
com.swiftkey.swiftkeyconfigurator.READCONFIG RECORD_AUDIO
com.google.android.c2dm.permission.RECEIVE
FOREGROUND_SERVICE

Table 7.1: The list of permissions requested by SwiftKey.

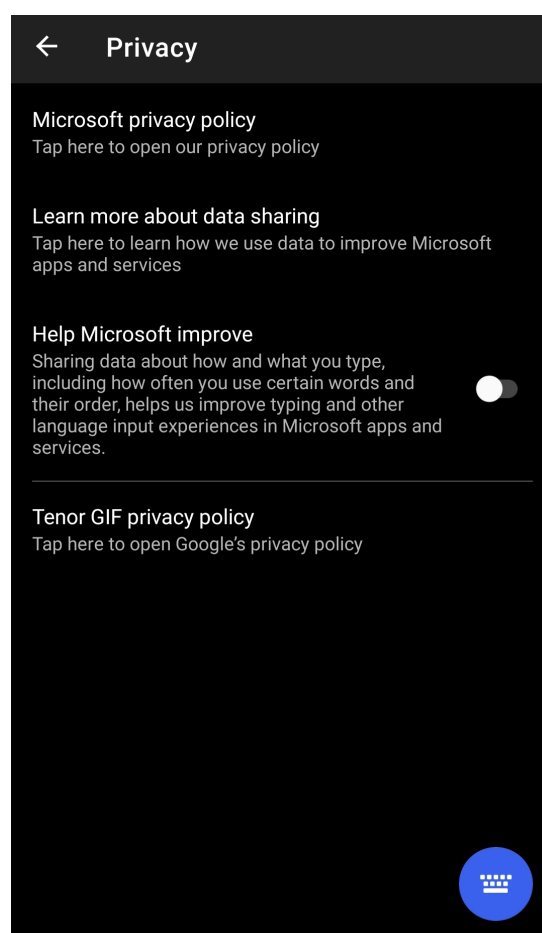
Despite the privacy statement being a generic one, Microsoft does provide additional information about some of its products under the "Product-specific details" section. To Microsoft's credit, this section does, in fact, include SwiftKey. It is claimed that SwiftKey uses data about typed words to learn a writing style and provide personalised autocorrection and predictive text. This is done by building a personalised language model, which learns about commonly used words and phrases and adapts to different scenarios, such as using specific applications or visiting websites. The model is claimed to "attempt to avoid collecting sensitive data" such as passwords or payment details. It is then described that SwiftKey does not "log, store or learn" from typing data unless the user has opted in. However, it is then stated that "de-identified" device data and usage statistics are always collected when using SwiftKey Services, and there is no opt-out option described. The term "usage statistics" is not expanded on in the privacy statement. However, some information is provided on a SwiftKey support page (45), where it is described as meta-level insights into what is typed on the keyboard. The lack of an opt-out option means that whether or not users opt-in to the personalisation of their typing experience, telemetry data is still collected about their interactions with the keyboard and shared with Microsoft's servers. Details are then provided in the privacy statement about SwiftKey Accounts and how language models are synced with them if users opt to log in. It is stated that when creating a SwiftKey Account, email and other "basic demographic" data are collected. It is then also described that users may opt-in to sharing snippets of their typing data and voice clips to help Microsoft improve their products and services. These snippets are claimed to be "de-identified" and not linked to any accounts, however, not much information is provided about how these snippets are generated.

## 7.4 Opting Out

There are multiple telemetry streams on SwiftKey, and it is not possible to opt out of all of them. Upon the initial launch, the application explicitly asks users for their consent to share data about how they type, as shown in Figure 7.1a. Even if users say no, this only turns off one stream of telemetry, which is otherwise sent to the `snippetdata.api.swiftkey` endpoint. This telemetry collection is also available as a setting called "Help Microsoft improve" under the Privacy tab in SwiftKey's settings, as shown in Figure 7.1b. What is not made clear to users is that telemetry is still collected regardless of their consent to this setting. Particularly usage statistics about their interactions with the application, sent to the `telemetry.api.swiftkey` endpoint. There is no way for users to opt out from this telemetry being collected and sent to back-end servers.



(a) The telemetry opt-in screen presented on fresh install.



(b) The privacy settings.

Figure 7.1: The telemetry options available on SwiftKey.



## 7.5 Experiment Results

When the keyboard application is launched, connections are almost instantly made to the `telemetry.api.swiftkey` endpoint, sending hardware information about the handset and configuration settings about the keyboard application. This includes details such as the handset's OS, model, manufacturer, architecture, CPU count, total RAM, display size and density, operator details, and a Google advertising ID, which has been shown to be a strong device identifier (39). The version of the SwiftKey application is also logged, along with the state of various settings, for example, autocorrect being turned on or the theme being used.

```
{ 'event':  
  { 'metadata':  
    { 'installId': <...>, 'appVersion': '7.8.3.5',  
      'timestamp': { 'utcTimestamp': 1647700369083, 'utcOffsetMins': 0 },  
      'vectorClock': { 'major': 4, 'minor': 1, 'order': 100 },  
      'deviceInfo': { 'os': { 'name': 'ANDROID', 'version': '11' },  
        'model': 'Pixel 2', 'manufacturer': 'Google',  
        'architecture': 'arm64-v8a', 'cpus': 8, 'totalRam': 3834560512,  
        'screenMetrics': { 'density': 420, 'width': 1080, 'height': 1794 },  
        'deviceId': '', 'operator': { 'name': '', 'country': 'ie' },  
        'locale': 'en_US', 'language': 'en', 'advertisingId': '123456789',  
        'accessibilityScreenReaderEnabled': False,  
        'pushNotificationId': '...', 'sdkVersion': '5.0.0.54'  
      }  
    }  
  }  
}
```

### 1. Messaging application (Typing) - WhatsApp

It was found that whenever text is entered through the keyboard, this is logged in extensive detail. The name of the application where the keyboard is tracked, along with the timestamp at which the keyboard was used and the duration. A field labelled as "typingStats" is also present within the same log. This field includes information about the total amount of tokens (words) entered, how many of these were predicted or typed in verbatim, and the total counts of characters entered and deleted are also tracked. Additionally, information such as predicted keystrokes or the amount of emojis entered is also present. It does not appear that SwiftKey tracks any indication of which characters were entered, only how many of them. Each time an experiment string was entered and sent, a log like this was created.

```

{'event':
  {'metadata':
    {'installId': <...>, 'appVersion': '7.8.3.5',
     'timestamp': {'utcTimestamp': 1648158169587, <...>},
     'vectorClock': {'major': 1, 'minor': 656, 'order': 100}},
    'application': 'com.whatsapp', 'durationMillis': 38002,
    'typingStats':
      {
        'totalTokensEntered': 9, 'tokensFlowed': 0,
        'tokensPredicted': 0, 'tokensCorrected': 0,
        'tokensVerbatim': 9, 'tokensPartial': 0,
        'netCharsEntered': 45, 'deletions': 0,
        'characterKeystrokes': 0, 'predictionKeystrokes': 0,
        'remainderKeystrokes': 0, 'predictionSumLength': 0,
        'typingDurationMillis': 10146, 'emojisEntered': 0,
        'totalTokensEnteredEdited': 0, 'tokensFlowedEdited': 0,
        'tokensPredictedEdited': 0, 'tokensCorrectedEdited': 0,
        'tokensVerbatimEdited': 0, 'tokensPartialEdited': 0
      } <...>
    }
  }
}

```

## 2. Browser (Typing) - Chrome

The results obtained were largely similar to those described in Experiment 1, suggesting that the application in which the keyboard is used does not affect the telemetry.

## 3. Messaging application (Voice) - WhatsApp

No "typingStats" logs about the inputs were observed when using the voice typing feature. This is likely because SwiftKey relies on Google's voice typing service for this feature. It also seems that the start and end times of the voice input are not tracked.

## 4. Messaging application (Gesture) - WhatsApp

The results obtained in this experiment are largely similar to those from Experiment 1. However, within the "typingStats" field, it is logged that the tokens and characters were "flowed", which is SwiftKey's naming for gesture or swipe typing.

```

{'event':
  <...>,
  'application': 'com.whatsapp', 'durationMillis': 15001,
  'typingStats': {'totalTokensEntered': 5, 'tokensFlowed': 5, <...>}
}

```

#### 5. Entering Password (Typing) - Pinterest

Neither the length of the email or the password were observed to be logged within the typing statistics. The "typingStats" entries were generated with the corresponding application names, but the number of characters entered and the remaining values were logged as 0.

#### 6. Entering Password (Clipboard Paste) - Pinterest

Similar results were observed to those described in Experiment 5.

#### 7. Other Language (Typing) - WhatsApp

Regardless of the language used, following the "typingStats" field, information is logged about the language details of the entered text. This includes how many and what languages were used, how many terms were entered per language, and how many tokens were shown as suggestions per each language.

```
{'event':
  {'metadata':
    {<...>,
      'application': 'com.whatsapp', 'durationMillis': 22718,
      'typingStats':
        {
          'totalTokensEntered': 8, 'tokensFlowed': 0,
          'tokensPredicted': 0, 'tokensCorrected': 1,
          'tokensVerbatim': 7, 'tokensPartial': 0,
          'netCharsEntered': 43, 'deletions': 2,
          <...>
          'typingDurationMillis': 9025, 'emojisEntered': 0,
          <...>
        },
        'languagesUsed': 1, 'termsPerLanguage': {'en_GB': 8},
        'tokensPerSource': {'en_GB/en_GB.lm': 2,
          'user/dynamic.lm': 6}, 'tokensShownPerSource':
        {'': 25, 'en_GB/en_GB.lm': 44, 'en_GB/en_GB_word_c.lm1': 5,
          'fr_FR/fr_FR_word_c.lm1': 1, 'user/dynamic.lm': 58,
          'fr_FR/fr_FR.lm': 25}, 'userHandle': 0
      }
  }
```

#### 8. Messaging application conversation (2 Phones Typing) - WhatsApp

This experiment was conducted to investigate whether it would be possible to link two handsets to be communicating with each other based on the logs generated by their keyboards. Both handsets generated logs similar to those described in Experiment 1

during a simulated conversation. Based on this, it can be concluded that it would be theoretically possible to link the handsets based on the timing information of when messages are sent. This concept is further expanded on in Chapter 9.

## 9. Settings

It was observed that SwiftKey keeps track of the individual settings screens that are visited and precisely at what time. This is done by logging the name of the current settings page and the name of the previous one. This means that all the interactions a user has with the settings can be traced back, and due to the timestamps, the amount of time spent on each screen can also be inferred.

```
{
  <...>, 'utcTimestamp': 1647700939979, <...>,
  'pageName': 'SETTINGS', 'prevPageName': 'LANGUAGE_SETTINGS',
  'pageOrigin': 'SETTINGS', 'id': '<...>'
}
{
  <...>, 'utcTimestamp': 1647700940539, <...>,
  'pageName': 'TYPING_CONSENT_SETTINGS', 'prevPageName': 'SETTINGS',
  'pageOrigin': 'SETTINGS', 'id': '<...>'
}
```

## 10. Privacy Policy

It is observed that the action of clicking on the privacy policy link available under the "Privacy" screen is also explicitly timestamped and logged.

```
{
  <...>, 'utcTimestamp': 1649617904115, <...>,
  'pageName': 'TYPING_CONSENT_SETTINGS', 'prevPageName': 'SETTINGS',
  'pageOrigin': 'SETTINGS', 'id': '<...>'
}
{
  <...>, 'utcTimestamp': 1649617914488, <...>
  'action': 'LINK_PRIVACY'
}
{
  <...>, 'utcTimestamp': 1649617915224, <...>,
  'pageName': 'TYPING_CONSENT_SETTINGS', 'id': '<...>'
}
```

## 8 AnySoftKeyboard

### 8.1 Application Overview

AnySoftKeyboard is an open-source Android keyboard that is widely regarded as one of the best privacy oriented keyboards available. It can be downloaded from the Play Store (6), where it has over a million installs, or from F-Droid, an installable catalogue of free and open-source software (15). It is highly customisable and supports multiple languages, gesture typing, dictionaries and word predictions, among other keyboard features.

### 8.2 Permissions

The keyboard does not require any special permissions in order to function. It is possible to decline all the permissions requested and still use the application as normal. The list of permissions that are asked for is shown in Table 8.1.

Android Permission
VIBRATE
READ_USER_DICTIONARY
WRITE_USER_DICTIONARY
READ_CONTACTS
WRITE_EXTERNAL_STORAGE

Table 8.1: The list of permissions requested by AnySoftKeyboard.

AnySoftKeyboard is transparent about why these permissions are requested and what they are used for. This is explained both in comments within the source code and on a dedicated wiki page (5). Granting the application with these permissions enhances some of its features and does not put the user's privacy at risk. Most notably, due to the lack of internet permissions, meaning that the application cannot send any data anywhere outside the device it is installed on.

## 8.3 Privacy Policy

The privacy policy (4) states that no personal information or data from the applications' storage is ever shared with anyone or transferred away from the mobile handset. The only data collected is typing behaviour, which is used to improve the application's dictionary and word prediction features. This data is stored locally and never leaves the device. Additionally, if granted permission, the application will also be able to access the contacts list from the device. The privacy policy is extremely clear and easy to read. It can also be easily accessed within the application and on the web.

## 8.4 Opting out

The keyboard application does not share any user telemetry data, and as such, no option for opting out is available. Users can choose to deny any permissions from being granted to the application. This prevents it from being able to create and update a user dictionary, read the device's contacts list and write to external storage to create backups.

## 8.5 Experiment Results

The experiments were performed for AnySoftKeyboard in the same way as for the other two previously mentioned keyboards. Expectedly though, no connections were observed to be made by the open-source application due to the lack of network permissions. As such, it can be concluded that AnySoftKeyboard lives up to its reputation of respecting user privacy and does not send any telemetry to back-end servers.

## 9 Evaluation

Similarly to previous work (35) (39) (37), this research focuses on the perspective of an average privacy-conscious user, who will reject any telemetry collecting options if prompted with the choice, but will otherwise continue with the default settings. Assuming this perspective provides a baseline for a privacy analysis, and it is possible that less aware users might witness higher amounts of data sharing. It is also important to outline that only a subset of the data is addressed when discussing the results and implications of observed data collection. This is because the omitted data was either repetitive or uninteresting and hence irrelevant to a discussion about user privacy.

### 9.1 Permissions

It is clear that applications such as AnySoftKeyboard can exist and perform functionally without special permissions. However, both Gboard and SwiftKey require many more system resources, largely due to their vast feature sets. As always, there is a trade-off here. For example, giving a keyboard application internet permissions allows it to support features such as retrieving and sending GIFs or syncing the user's personal dictionary with the cloud. On the other hand, permitting an application to access the network at any time allows it to communicate with its back-end servers and share telemetry about its users.

The primary issue is that users often have minimal control over which permissions they can reject. On top of that, the way in which an application uses its permissions is almost always very ambiguous (32) (47). Giving users more control over the permissions of their applications would allow them to opt out of specific features which they are not interested in, at the benefit of preserving their privacy. Apple has found a solution for this problem on iOS by limiting the permissions of third-party keyboards, including internet access. For a keyboard to get all of its permissions, a user needs to grant it "Full Access" explicitly, and Apple warns its users about the potential threats caused by doing this in a warning message. Implementing such access control on Android would be greatly beneficial to giving users more control over their privacy.

## 9.2 Transparency

As described previously, the privacy policies of Gboard and SwiftKey do not provide users with much clarity about how their data is handled when using these applications. Admittedly, Microsoft's statement does have a section specifically for its keyboard; however, ambiguous terms such as "usage statistics" are used without further descriptions. In the case of SwiftKey, this is especially an issue considering there is no opt-out available, and this data is continuously collected. Gboard does offer an opt-out of its data sharing, though it is somewhat unclear as to what data is collected in the first place. Therefore, users cannot make informed decisions about their consent to this telemetry because they are not well-informed. AnySoftKeyboard, on the other hand, shows that the choice of being transparent or not is one entirely made by the application vendor. This keyboard is open-source, and it also provides explicit commentary about how its permissions are used and for what purposes. Additionally, the privacy policy provided is extremely clear and focused and is simple to access and understand.

It would be beneficial to users if the privacy policies provided by each application were specific to the keyboards themselves. They should also clearly outline what data is collected and shared, what the purpose of the data collection is, and how, where, and for how long the data is stored. In order to view the privacy policies of the applications studied in this work, it is necessary to navigate several screens within the application's settings and then agree to Chrome's terms of service since the policies open up as a link inside the browser. By providing a keyboard specific policy, it would be possible to make it accessible within the application itself without requiring a browser to view it, making it a more straightforward process for users to access. It would also make the policy more focused and easier to read as it would not have to be generic and vague to cover all of the company's services.

## 9.3 Data Collection

Based on the experimental results observed, it is evident that both Gboard and SwiftKey collect a large amount of telemetry about users' interactions with these applications. It was also observed that neither of the applications sends aggregated or histogram data based on locally accumulated usage statistics. Instead, every interaction that a user has with the keyboard is explicitly tracked and logged in a separate log entry. This naturally raises the question of whether it is necessary to collect all of this data. What is the purpose of this data collection? Is it excessive for the use case? AnySoftKeyboard, which has been shown not to share any user telemetry data, demonstrates that excessive data collection is not a requirement for developing a fully functional keyboard and is instead a choice made by the developers. While it can be useful to collect telemetry to help with the application's stability on different hardware or catch performance bugs, etc., this data does not need to be



extremely specific or linked to an individual. It has been shown that if the data is not uniquely identifying and is common to a high number of handsets, then it does not pose substantial privacy concerns (52). However, it has been observed that the logs shared by Gboard and SwiftKey can both be linked back to a handset and potentially an individual's identity due to the presence of unique identifiers.

Gboard does not require users to be logged into a Google account to use the application. However, an account is a requirement to download anything from the Play Store and get more utility from the device. This means that most Android users have likely logged into their Google account on the device at some point, thus linking their handset to the account. The connections made to the `play.googleapis.com/log/batch` endpoint by Gboard are tagged with an Android ID, which is linked to the handset's Google account, which can often mean a user's real identity (35). This is because a user's Google account has personally identifiable information linked to it, such as a phone number, which is required to set up the account, or credit card details used in payments on the Play Store or through Google Pay. Likewise, SwiftKey is also guilty of including a unique identifier within the logs sent to the `telemetry.api.swiftkey` endpoint. The Google Advertising ID is included inside the device information field. The Advertising ID can be reset, either manually by a user or through a factory reset of the handset. However, this rarely occurs in practice, and the field acts as a strong device identifier. In fact, it is listed as a unique identifier in Google's privacy policy (21). Telemetry data being tagged with a unique identifier makes it possible to link other data together and connect it all to a specific handset. This data can potentially be sensitive and reveal other information about the handset owner based on their usage patterns. It should be noted that there already exist commercial services which, provided the Google Advertising ID of a handset, can identify a user's personal information such as their name, address, or email (55). Data being marked with a unique identifier also makes it possible for two separate data collectors, such as Microsoft and Google, to link their data together about a specific handset and potentially target it with personalised ads based on their combined information. (Note: it is not claimed that this occurs in practice, but simply that it would theoretically be possible.)

While both keyboards collect and share a lot of data, no evidence was found to suggest that either keyboard tracks what words or characters are entered, although metadata about this content is logged. This includes information such as the length of individual words and how the content was entered - whether that was through text, gesture typing or voice input. The languages used when entering the content are logged, and in SwiftKey's case, how many words were entered per each language. Both keyboards also log how many characters were entered in total, how many were deleted, whether any of the words were suggested or autocorrected, and the time it took to enter the text. The presence or frequency of individual characters does not appear to be shared. While this information is very intrusive, on its own it poses little privacy concern. However, this data is not logged in isolation and is accompanied by very

precise timestamps and the names of the applications in which the keyboards were used. The timing information of when a user opens specific applications and how long they use them can potentially be sensitive information. For example, many applications for tracking menstrual cycles offer a feature of writing down notes or securing the app behind a password. Both of these features require the user to enter input with their keyboard, and therefore this activity is logged. This can disclose the user's sex and potentially reveal private personal information about their menstrual cycle. Dating apps are another example of how timestamped usage of applications can potentially be sensitive and revealing. If it is observed that a user spends a significant amount of time within specific dating apps, they likely have a personal motivation to do so. Depending on the application, this can potentially reveal the user's sexuality or age range.

Considering the sensitivity of timing information, an investigation was made to determine whether it would be possible to identify that two parties are interacting with each other based on their keyboard logs. This was performed by holding a WhatsApp conversation between two rooted devices, described previously as Experiment 8. There are several fields in the logs which could be used to help with identifying someone, such as hardware information about their device, what application they are using to communicate and what languages they are typing in; however, as described above, there is no need to try to interpret that information due to the presence of unique identifiers. The logs from each device can only identify the sender because the keyboard does not know who the receiver is and has no way to find out. Instead, the timing information of when messages are entered within an application can potentially be used to infer that two handsets are communicating. This data can be graphed and analysed for any patterns that emerge. A sample conversation between two devices that took place for twenty minutes is illustrated in Figure 9.1, where it can be seen that the handsets generally start and stop sending messages around the same time. Naturally, the data will not always perfectly align, as replies to messages are not always immediate and may occur at different rates depending on how busy a user is on any given day. A user may also be involved in conversations with more than one person at a time while using the same application. These messages will appear on the graph as noise. Finding patterns in the data does not guarantee communication between the two parties. However, analysing more data over time will lead to a higher degree of confidence in correlating the handsets to be communicating.

Both Gboard and SwiftKey also track a user's activity within the keyboard applications, which does not include typing. As described previously in Chapter 5, Gboard tracks user activity with keyboard events. Such events were observed for opening and closing the settings, although nothing was shared about which screens were visited within that time. SwiftKey, on the other hand, collects and shares data about which screens are visited, precisely at what timestamps, and how the users arrived at their current screen. Accessing the privacy policy is also an activity logged by Microsoft's keyboard. This was not observed to be done by Gboard. While

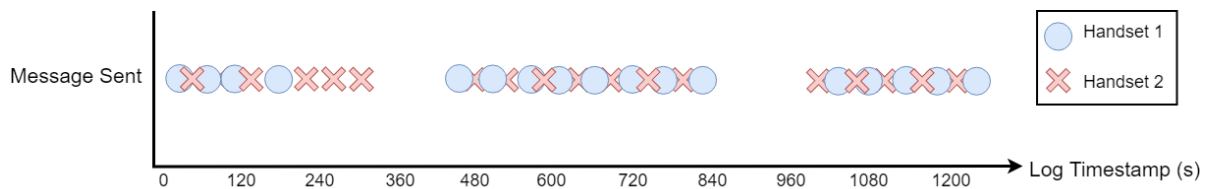


Figure 9.1: Example of the timestamp information of a WhatsApp conversation between two handsets, with the timestamps scaled up to seconds from milliseconds, and offset so that the conversation starts at timestamp 0.

this data in itself is not very revealing, it has to be questioned why Microsoft collects such excessively specific telemetry. An alternative to this could be, for example, to track how many times each screen was visited within a specific time window, e.g. the last hour. Aggregating data like this makes it less specific and intrusive yet still portrays the same inherent value of monitoring how frequently the settings screens are visited in comparison to each other.

The keyboards studied also have different approaches to storing user dictionary data. Over time as the keyboards are used, if permitted, the applications build up a dictionary of custom vocabulary, which provides a more personalised experience. Gboard and AnySoftKeyboard store everything locally and show no evidence of this data ever leaving the handset. However, SwiftKey offers the feature of syncing this data across devices by storing it in the cloud if signed in to an account. This naturally puts the data in a vulnerable state and raises security concerns about what could happen if a bad actor got their hands on a target's personalised dictionary. These concerns are not unrealistic, and in fact, the cloud syncing feature already had to be suspended for a period of time in 2016 (9), after users reported that their keyboards were suggesting phrases in languages they have never used and even personal information of other users such as their email addresses or phone numbers. This data leak occurred due to a bug in Microsoft's synchronisation code, and there is no guarantee that it will not happen again in the future.

# 10 Conclusion

## 10.1 Overview

This dissertation set out to tackle and shed light on the opaque nature of the data collection done by popular Android keyboards, namely Gboard, SwiftKey and AnySoftKeyboard. This was performed by collecting the network traffic generated by each application and doing a large amount of non-trivial reverse engineering work to decrypt and decode the content of these connections. The decoded data was analysed to see whether it contained any sensitive information. It was indeed found that the telemetry generated by both Gboard and SwiftKey could be linked to a specific handset and, in some cases, the user's real identity due to the presence of unique identifiers such as the Android ID and the Google Advertising ID.

While Gboard provides an option to opt out of this telemetry collection, it is not transparent about what data is collected and shared with its back-end servers. On the other hand, SwiftKey does not provide an option for users to opt out and continuously collects telemetry about the users' interactions with the keyboard. The "usage statistics" collected by both proprietary keyboards have been found to include specific hardware information about the device using the application, information about the length of individual words entered and their input method, how long it took the user to enter the text, how many characters were entered or deleted as well as what languages were used. The name of the application in which the keyboard was used is also collected. It should be stated that neither Gboard nor SwiftKey collect aggregated or histogram data and instead log every user interaction in a separate, timestamped log entry. The Microsoft keyboard even tracks individual screens visited by the user within the applications settings page. Neither keyboard has been observed to collect the input content or track the frequency of individual characters. The data that is collected, though, is quite intrusive and has been shown in Chapter 9 that it can be potentially used to reveal certain personality traits about a user. AnySoftKeyboard, the open-source alternative which was studied, does not collect or share any data at all. Proving that excessive telemetry collection is not a requirement to develop a fully functional keyboard and is instead a choice made by the developers.

Ultimately, it is up to each user to decide how much they value their privacy and make a choice

about their consent to the collection of telemetry data done by the studied applications. The findings from this work will allow users to no longer be left in the dark about what data is collected and shared when typing on their keyboards. Instead, it will enable them to make an informed decision, allowing them to be in charge of their privacy.

## 10.2 Future Work

The work presented in this report outlines how three different Android keyboard applications have been analysed from a user privacy perspective. Naturally, various applications will differ in their security approaches; however, the experimental setup used for data collection and the reverse engineering techniques can also be applied to any other mobile application. The results of this work, such as the reconstructed Gboard protobuf definition file, as well as the tools used for reverse engineering, have been published to GitHub (31) to allow others to continue from where this work has left off.

As noted previously, this work does not present an exhaustive reverse engineering of Gboard. Therefore, this could merit further investigation that could build on top of the presented results to gain a deeper understanding of everything logged by the keyboard application. Gboard also employs federated learning in order to learn new words and phrases (25). This feature could be further investigated to verify whether any sensitive information is shared as part of "improving" Gboard. With regard to SwiftKey, future work could address the 'snippetdata' connections that were observed when a user opted into helping "Microsoft improve" (44), as these could not be investigated during the time frame of this work.

Future work in keyboard privacy could investigate Samsung's Android keyboard or look into Apple's keyboard on iOS to provide a point of comparison across the two operating systems. Apple has the reputation of placing a greater emphasis on privacy than Google, yet research (33) shows that there is, in fact, little difference between the privacy of applications regardless of the operating system. It would be interesting to see if this also applies to keyboards.

Finally, it should be noted that these are live, widely deployed applications that are constantly updated. It is essential that future versions of these applications do not neglect their users' privacy and that any changes that could affect this are clearly communicated. If such changes are made, they will merit further investigation to fully understand their impact.

# Bibliography

- [1] Android (2021), 'Android enterprise security paper'. Accessed December 2021.  
URL: [https://source.android.com/security/reports/Google\\_Android\\_Enterprise\\_Security\\_Whitepaper\\_2020.pdf](https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2020.pdf)
- [2] Android (2022a), 'Android activity documentation'. Accessed February 2022.  
URL: [https://developer.android.com/reference/android/app/Activity#onStart\(\)](https://developer.android.com/reference/android/app/Activity#onStart())
- [3] Android (2022b), 'Android debug bridge'. Accessed March 2022.  
URL: <https://developer.android.com/studio/command-line/adb>
- [4] AnySoftKeyboard (2017), 'Anysoftkeyboard privacy policy'. Accessed February 2022.  
URL: <https://anysoftkeyboard.github.io/privacy-policy/>
- [5] AnySoftKeyboard (2018), 'Why does anysoftkeyboard require extra permissions'. Accessed February 2022.  
URL: <https://github.com/AnySoftKeyboard/AnySoftKeyboard/wiki/Why-Does-AnySoftKeyboard-Requires-Extra-Permissions>
- [6] AnySoftKeyboard (2022), 'Anysoftkeyboard play store page'. Accessed February 2022.  
URL: [https://play.google.com/store/apps/details?id=com.menny.android.anysoftkeyboard&hl=en\\_IE&gl=US](https://play.google.com/store/apps/details?id=com.menny.android.anysoftkeyboard&hl=en_IE&gl=US)
- [7] Apache (2021), 'Avro documentation'. Accessed March 2022.  
URL: <https://avro.apache.org/docs/current/>
- [8] Apache (n.d.), 'Apache avro api documentation'. Accessed February 2022.  
URL: <https://avro.apache.org/docs/current/api/java/org/apache/avro/file/DataFileStream.html>
- [9] Authority, A. (2016), 'Swiftkey suspends cloud syncing after data leak'. Accessed April 2022.  
URL: <https://www.androidauthority.com/swiftkey-suspends-service-data-leak-706680/>

- [10] Barbon, G., Cortesi, A., Ferrara, P., Pistoia, M. and Tripp, O. (n.d.), Privacy analysis of android apps: Implicit flows and quantitative analysis, *in* 'Computer Information Systems and Industrial Management', Springer-Verlag, Berlin, Heidelberg, p. 3–23.
- [11] Barth, S. and de Jong, M. D. (2017), 'The privacy paradox – investigating discrepancies between expressed privacy concerns and actual online behavior', *Telematics and Informatics* **34**(7), 1038–1058.
- [12] Buffers, P. (n.d.), 'Protocol compiler'. Accessed October 2021.  
URL: <https://github.com/protocolbuffers/protobuf>
- [13] Cho, J., Cho, G. and Kim, H. (2015), Keyboard or keylogger?: A security analysis of third-party keyboards on android, *in* '2015 13th Annual Conference on Privacy, Security and Trust (PST)', pp. 173–176.
- [14] ElderDrivers (n.d.), 'Edxposed'. Accessed October 2021.  
URL: <https://github.com/ElderDrivers/EdXposed>
- [15] F-Droid (2022), 'Anysoftkeyboard'. Accessed February 2022.  
URL: <https://f-droid.org/packages/com.menny.android.anysoftkeyboard/>
- [16] Frida (n.d.), 'Frida documentation'. Accessed February 2022.  
URL: <https://frida.re/>
- [17] Gibler, C., Crussell, J., Erickson, J. and Chen, H. (2012), Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale, *in* 'TRUST'.
- [18] Google (2021a), 'Gboard app store page'. Accessed October 2021.  
URL: <https://apps.apple.com/us/app/gboard-the-google-keyboard/id1091700242>
- [19] Google (2021b), 'Gboard play store page'. Accessed October 2021.  
URL: <https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin>
- [20] Google (2021c), 'Google i/o 2021'. Accessed November 2021.  
URL: <https://io.google/2021/?lng=en>
- [21] Google (2022a), 'Google privacy policy'. Accessed January 2022.  
URL: <https://policies.google.com/privacy>
- [22] Google (2022b), 'Protocol buffers message structure'. Accessed March 2022.  
URL: <https://developers.google.com/protocol-buffers/docs/encoding#structure>

- [23] Google (2022c), 'Protocol buffers overview'. Accessed March 2022.  
URL: <https://developers.google.com/protocol-buffers/docs/overview>
- [24] Google (n.d.a), 'Compact protobuf comments'. Accessed December 2021.  
URL: <https://cs.android.com/android/platform/superproject/+master:external/protobuf/java/core/src/main/java/com/google/protobuf/RawMessageInfo.java>
- [25] Google (n.d.b), 'Learn how gboard gets better'. Accessed April 2022.  
URL: <https://support.google.com/gboard/answer/9334583?hl=en>
- [26] Google (n.d.c), 'Quic protocol documentation'. Accessed January 2022.  
URL: <https://www.chromium.org/quic/>
- [27] Graux, P., Lalande, J.-F. and Tong, V. V. T. (2019), Obfuscated android application development, in 'Proceedings of the Third Central European Cybersecurity Conference', CECC 2019, Association for Computing Machinery.
- [28] iBotPeaches (2022), 'Apktool'. Accessed January 2022.  
URL: <https://ibotpeaches.github.io/Apktool/>
- [29] Ito, K., Hasegawa, H., Yamaguchi, Y. and Shimada, H. (2018), Detecting privacy information abuse by android apps from api call logs, in A. Inomata and K. Yasuda, eds, 'Advances in Information and Computer Security', Springer International Publishing, Cham, pp. 143–157.
- [30] Jin, H., Liu, M., Dodhia, K., Li, Y., Srivastava, G., Fredrikson, M., Agarwal, Y. and Hong, J. I. (2018), 'Why are they collecting my data? inferring the purposes of network traffic in mobile apps', *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2(4).
- [31] kamilprz (n.d.), 'Android keyboard privacy investigation tools'. Accessed April 2022.  
URL: <https://github.com/kamilprz/android-keyboard-privacy>
- [32] Khatoon, A. and Corcoran, P. (2017), Android permission system and user privacy — a review of concept and approaches, in '2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)', pp. 153–158.
- [33] Kollnig, K., Shuba, A., Binns, R., Kleek, M. V. and Shadbolt, N. (2022), 'Are iphones really better for privacy? a comparative study of ios and android apps', *Proceedings on Privacy Enhancing Technologies* 2022, 6 – 24.
- [34] Leith, D. J. (2021), Mobile handset privacy: Measuring the data ios and android send to apple and google, in 'SecureComm'.



- [35] Leith, D. J. (2022), What data do the google dialer and messages apps on android send to google?, Technical report, Trinity College Dublin, School of Computer Science & Statistics.
- [36] Leith, D. J. (n.d.), 'Inspecting phone network connections'. Accessed April 2022.  
URL: <https://github.com/doug-leith/cydia#android>
- [37] Leith, D. J. and Farrel, S. (2020), Coronavirus contact tracing app privacy: What data is shared by the singapore opentrace app?, in 'SecureComm'.
- [38] Leith, D. J. and Farrell, S. (2021), 'Contact tracing app privacy: What data is shared by europe's gaen contact tracing apps', *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications* pp. 1–10.
- [39] Leith, D. J., Liu, H. and Patras, P. (2021), Android mobile os snooping by samsung, xiaomi, huawei and realme handsets, Technical report, Trinity College Dublin, University of Edinburgh.
- [40] Ltd., R. P. (n.d.), 'Raspberry pi documentation'. Accessed November 2021.  
URL: <https://www.raspberrypi.com/documentation/computers/configuration.html>
- [41] Mann, C. and Starostin, A. (2012), A framework for static detection of privacy leaks in android applications, in 'Proceedings of the 27th Annual ACM Symposium on Applied Computing', SAC '12, Association for Computing Machinery, New York, NY, USA, p. 1457–1462.
- [42] Microsoft (2022a), 'Microsoft privacy statement'. Accessed March 2022.  
URL: <https://privacy.microsoft.com/en-gb/privacystatement>
- [43] Microsoft (2022b), 'Swiftkey play store page'. Accessed February 2022.  
URL: <https://play.google.com/store/apps/details?id=com.touchtype.swiftkey>
- [44] Microsoft (n.d.a), 'Sharing your typing data faq'. Accessed April 2022.  
URL: <https://support.swiftkey.com/hc/en-us/articles/115001737845>
- [45] Microsoft (n.d.b), 'What data is collected / sent while using microsoft swiftkey keyboard?'. Accessed April 2022.  
URL: <https://support.swiftkey.com/hc/en-us/articles/201454572-What-data-is-collected-sent-while-using-Microsoft-SwiftKey-Keybaord->
- [46] mitmproxy (n.d.), 'Mitmproxy documentation'. Accessed November 2021.  
URL: <https://docs.mitmproxy.org/stable/>

- [47] Mylonas, A., Theoharidou, M. and Gritzalis, D. (2014), Assessing privacy risks in android: A user-centric approach, *in* T. Bauer, J. Großmann, F. Seehusen, K. Stølen and M.-F. Wendland, eds, 'Risk Assessment and Risk-Driven Testing', Springer International Publishing, Cham, pp. 21–37.
- [48] OWASP (n.d.), 'Certificate and public key pinning'. Accessed March 2022.  
**URL:** [https://owasp.org/www-community/controls/Certificate\\_and\\_Public\\_Key\\_Pinning](https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning)
- [49] RikkaApps (n.d.), 'Riru'. Accessed October 2021.  
**URL:** <https://github.com/RikkaApps/Riru>
- [50] skylot (n.d.), 'jadx decompiler'. Accessed October 2021.  
**URL:** <https://github.com/skylot/jadx>
- [51] Square (2022), 'Okhttp'. Accessed January 2022.  
**URL:** <https://square.github.io/okhttp/>
- [52] Sweeney, L. (2002), 'K-anonymity: A model for protecting privacy', *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* **10**(5), 557–570.
- [53] topjohnwu (n.d.), 'Magisk'. Accessed October 2021.  
**URL:** <https://github.com/topjohnwu/Magisk>
- [54] Verderame, L., Caputo, D., Romdhana, A. and Merlo, A. (2020), 'On the (un)reliability of privacy policies in android apps', *CoRR* **abs/2004.08559**.
- [55] Vice (2021), 'Inside the industry that unmask people at scale'. Accessed April 2022.  
**URL:** <https://www.vice.com/en/article/epnmvz/industry-unmasks-at-scale-maid-to-pii>
- [56] Yen, T.-F., Xie, Y., Yu, F., Yu, R. P. and Abadi, M. (2012), Host fingerprinting and tracking on the web: Privacy and security implications, *in* 'NDSS'.