JISKA (Jurnal Informatika Sunan Kalijaga)
Vol. 10, No. 1, JANUARY, 2025, Pp. 122 – 138
ISSN: 2527 – 5836 (print) | 2528 – 0074 (online)

Android Malware Threats: A Strengthened Reverse Engineering Approach to Forensic Analysis

Ridho Surya Kusuma (1)*, **M Dirga Purnomo Putra** (2)
Department of Informatics, Universitas Siber Muhammadiyah, Yogyakarta, Indonesia
e-mail : {ridhosuryakusuma, dirga20220100042}@sibermu.ac.id
* Corresponding author.

This article was submitted on 29 August 2024, revised on 30 January 2025, accepted on 30 January 2025, and published on 31 January 2025.

Abstract

The widespread adoption of Android devices has rendered them a primary target for malware attacks, resulting in substantial financial losses and significant breaches of user privacy. Malware can exploit system vulnerabilities to execute unauthorized premium SMS transactions, exfiltrate sensitive data, and install additional malicious applications. Conventional detection methodologies, such as static and dynamic analysis, often prove inadequate in identifying deeply embedded malicious behaviors. This study introduces a systematic reverse engineering framework for analysing suspicious Android applications. In contrast to traditional approaches, the proposed methodology comprises six distinct stages: initialization, decompilation, static analysis, code reversal, behavioral analysis, and reporting. This structured process facilitates a comprehensive examination of an application's internal mechanisms, enabling the identification of concealed malware functionalities. The findings of this study demonstrate that the proposed method attains an overall effectiveness of 84.3%, surpassing conventional static and dynamic analysis techniques. Furthermore, this research generates a detailed list of files containing specific malware indicators, thereby enhancing the effectiveness of future malware detection and prevention systems. These results underscore the efficacy of reverse engineering as a critical tool for understanding and mitigating sophisticated Android malware threats.

Keywords: Malware Android, Reverse Engineering, Android Security, Digital Forensics, Cybersecurity

Abstrak

Adopsi perangkat Android yang meluas telah menjadikannya target utama serangan malware, yang mengakibatkan kerugian finansial yang besar dan pelanggaran signifikan terhadap privasi pengguna. Malware dapat mengeksploitasi kerentanan sistem untuk melakukan transaksi SMS premium yang tidak sah, mengeksfiltrasi data sensitif, dan memasang aplikasi berbahaya tambahan. Metodologi deteksi konvensional, seperti analisis statis dan dinamis, sering kali terbukti tidak memadai dalam mengidentifikasi perilaku berbahaya yang tertanam dalam. Penelitian ini memperkenalkan kerangka kerja reverse engineering yang sistematis untuk analisis aplikasi Android yang mencurigakan. Berbeda dengan pendekatan tradisional, metodologi yang diusulkan terdiri dari enam tahap yang berbeda: Inisialisasi, dekompilasi, analisis statis, pembalikan kode, analisis perilaku, dan pelaporan. Proses terstruktur ini memfasilitasi pemeriksaan komprehensif terhadap mekanisme internal aplikasi, memungkinkan identifikasi fungsi malware yang tersembunyi. Temuan penelitian ini menunjukkan bahwa metode yang diusulkan mencapai efektivitas keseluruhan sebesar 84,3%, melampaui teknik analisis statis dan dinamis konvensional. Selain itu, penelitian ini menghasilkan daftar file terperinci yang berisi indikator malware tertentu, sehingga berkontribusi pada peningkatan sistem pendeteksian dan pencegahan malware di masa mendatang. Hasil ini menggarisbawahi keampuhan reverse engineering sebagai alat penting untuk memahami dan memitigasi ancaman malware Android yang canggih.

Kata Kunci: Malware Android, *Reverse Engineering*, *Android Security*, *Digital Forensic*, *Cybersecurity*

1. INTRODUCTION

In cybersecurity, the prevalence of Android malware attacks represents a significant and escalating threat to the security of users, organizations, and the integrity of digital ecosystems. While previous studies have extensively highlighted (Manzil & Naik, 2023; Qamar et al., 2019), limited attention has been given to integrating advanced detection techniques with scalable forensic methods to address these challenges effectively. This research aims to bridge that gap by proposing a unified approach that examines the intricacies of Android malware behavior and develops enhanced methodologies for malware detection and forensic investigation (Umar et al., 2021b).

Unlike earlier works that primarily focused on specific malware categories, such as banking malware or SMS malware, this study presents a comprehensive approach that combines static and dynamic analysis methods to detect disguised and stealthy malware. Previous studies, such as those by Joseph Raymond and Jeberson Joseph Raymond & Jeberson Retna Raj (2023) and Liu et al. (2023), have explored the application of machine learning and deep learning in isolation. In contrast, this research integrates these approaches with novel forensic techniques, such as Just-in-Time Memory Forensics (JIT-MF), enabling effective real-time evidence collection during malware incidents (Bellizzi et al., 2022).

A key distinction of this study lies in its dual-layered methodology. While static analysis has proven effective for identifying vulnerabilities and data leaks in Android applications (Almomani et al., 2022), dynamic analysis is employed here to address runtime behaviors and the limitations of static approaches. This combination enhances malware detection capabilities and establishes a robust framework for defense mechanisms, particularly for Android devices integrated with IoT applications—a domain identified as particularly vulnerable (Ashawa & Morris, 2021).

Researchers have explored innovative approaches such as deep learning, machine learning, and dynamic analysis technologies to address the challenges posed by Android malware. For instance, Alkahtani & Aldhyani (2022) demonstrated that deep learning models significantly enhance malware detection accuracy, particularly with large-scale datasets. Similarly, Ye et al. (2022) highlighted how machine learning algorithms effectively identify previously unseen malware variants, reducing the risk of zero-day attacks. However, while these methodologies represent critical advancements, they often fail to address the challenges of real-time detection and evidence collection—an issue that this study directly tackles.

Furthermore, the emergence of disguised Android malware employing advanced evasion techniques has underscored the need for hybrid detection approaches. Elsersy et al. (2022) proposed frameworks that integrate static and dynamic analysis to overcome the limitations of individual methods. Building on this, Bellizzi et al. (2022) introduced JIT-MF as an innovative solution for collecting volatile memory evidence during active malware incidents. This research further expands upon these contributions by integrating JIT-MF with scalable detection frameworks, ensuring timely and efficient forensic investigations.

Despite significant advancements, a critical gap persists in integrating existing techniques into a unified framework that improves detection capabilities and ensures scalability and efficiency in forensic practices. This study addresses this gap by proposing a cohesive methodology that enhances both malware detection and forensic investigation processes, thereby strengthening the security of Android devices and the sensitive data they store. Rather than comparing the efficacy of detection methods—such as dynamic analysis or machine learning—this research focuses on identifying behavioral patterns and deriving forensic insights through reverse engineering techniques.

2. METHODS

The research method in this study uses a reverse engineering approach. The approach is a fundamental technique in Android malware forensics, which enables the extraction of digital evidence from malicious apps (Qiu et al., 2019). This process is crucial for uncovering how malware works, identifying its behaviour, and extracting valuable insights that can help investigate and mitigate Android malware attacks (Joseph Raymond & Jeberson Retna Raj, 2023). Reverse engineering is essential to overcome the challenges posed by sophisticated Android malware, including obfuscation techniques and encryption technologies designed to evade detection and Analysis (Ye et al., 2022) and (Urooj et al., 2022).

The flowchart below illustrates the sequential steps involved in the reverse engineering methodology for analyzing Android malware in digital forensics, as shown in Figure 1. Figure 1 describes the stages of the reverse engineering methodology in Android malware forensics, which consists of six main steps:

a) The initialization, which is the first step in acquiring and sampling Android malware, usually in the form of APK files (Lubuva et al., 2019).
b) Decompilation is decompiling the APK file to convert the binary code into a human-readable format.
c) Static analysis is a type of analysis that examines decompiled code to identify malicious behavior, such as data exfiltration, privilege escalation, and communication with command and control servers (Bhandari & Jusas, 2020b).
d) Code Reversing is an analysis that delves deeper into decompiled code through code reversal techniques to trace execution flow, identify encryption methods, and uncover obfuscation techniques used by malware to avoid detection (Mastino et al., 2022).
e) Behavioural Analysis is a comprehensive behavioural analysis that outlines the actions performed by the malware, including file modification, network communication, and system interaction (Serketzis et al., 2019).
f) Reporting is the process of documenting the results and pattern recognition of a detailed report that covers the malware’s characteristics, behavior, and potential impact (Bhandari & Jusas, 2020a). This report is significant evidence for further investigation and mitigation efforts (Bartliff et al., 2020; Kusuma, 2023).

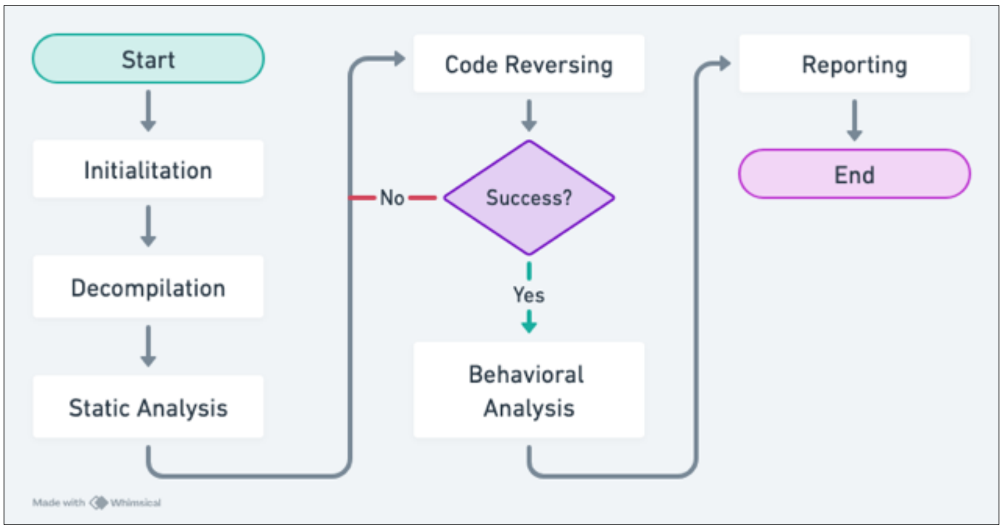


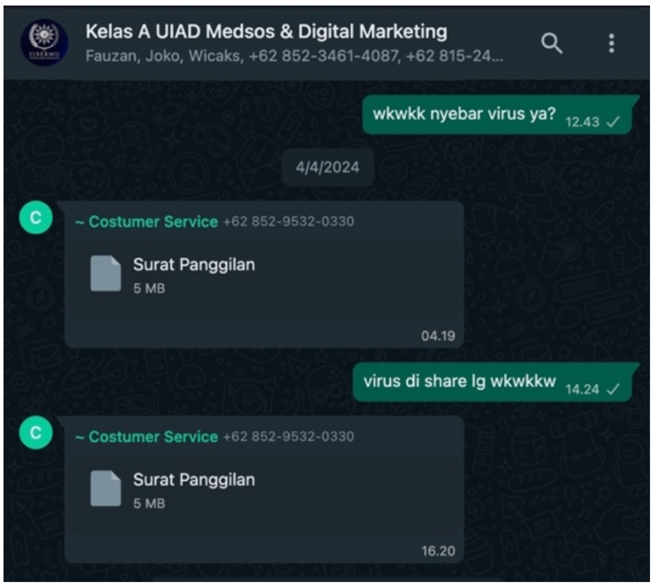
Figure 1 Flowchart of Reverse Engineering Method

A structured forensic methodology can effectively dissect Android malware, thus improving the ability to combat evolving cyber threats (Umar et al., 2021a). The following details the use of software in this research, specifically web applications, as outlined in Table 1. Table 1 shows that three web application-based tools are used to support the analysis and investigation of Android malware. The first tool is Decompiler (www.decompiler.com), which decompiles APK files into source code for further analysis. With this capability, Decompiler makes it easier to understand the structure and behavior of the analyzed application, particularly in identifying potentially suspicious activity.

Table 1 Web Application-Based Tools

Software	Description	Main Function	URL
Decompiler	A web-based tool used to analyze and decompile executable files.	Decompile Android application (APK) files into source code for further analysis.	www.decompiler.com
Metadefender	A cloud-based multi-scan platform that uses various antivirus engines for analysis.	Detects malware, vulnerabilities, and potential threats in APK files and other formats.	www.metadefender.com
Koodous	A community-based service that combines APK analytics with crowdsourced intelligence.	Detects malware in Android apps and provides deeper analysis of APK security.	www.koodous.com

The second tool is Metadefender (metadefender.com), a cloud-based multi-scan platform that uses various antivirus engines to detect malware and vulnerabilities in files. With its cloud-based approach, Metadefender enables more comprehensive detection by utilizing technologies from various security providers, thereby increasing the accuracy of the analysis. The third tool is Koodous (koodous.com), a community-based platform that offers security analysis for APK files. Koodous combines technical analysis with crowdsourced data from the security community, providing additional insights into malware and other security threats. The platform is particularly useful in detecting malicious apps that other tools may not identify, thanks to active community contributions.


Kelas A UIAD Medsos & Digital Marketing
Fauzan, Joko, Wicaks, +62 852-3461-4087, +62 815-24...

wkwkk nyebar virus ya? 12.43 ✓

4/4/2024

~ Costumer Service +62 852-9532-0330
Surat Panggilan
5 MB
04.19

virus di share lg wkwkkw 14.24 ✓

~ Costumer Service +62 852-9532-0330
Surat Panggilan
5 MB
16.20

Figure 2 Deployment of WhatsApp.apk

These three tools are used synergistically to ensure that Android malware analysis is thorough, whether through decompilation, antivirus-based detection, or community-based insights. The combination of these tools provides a strong foundation for investigating and mitigating security threats on the Android platform. The case study in this research is based on a real-world case from the course WhatsApp group and the use of datasets from Kaggle, sourced from URLs: https://www.kaggle.com/datasets/saurabhshahane/android-malware-dataset, to measure the effectiveness of this research method. The chronology begins when one of the numbers in the group with the pseudonym 'Customer Service' sends a file, as shown in Figure 2. Based on Figure

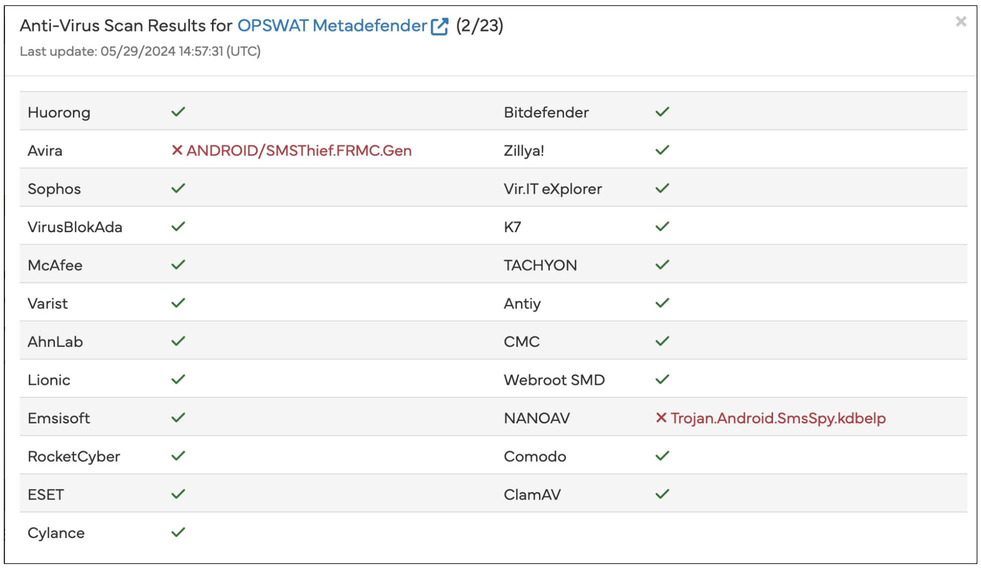
2, the appearance of a file sent via WhatsApp with a name resembling an official document, namely 'Surat Panggilan.apk'. This naming can trick other group members into opening the file and running it directly. This file has a '.APK' format, indicating an unknown and suspicious Android program.

3. RESULTS AND DISCUSSION

This research involves digital forensics and reverse engineering of Android APK files originating from WhatsApp groups. The file named 'Surat Panggilan.APK' looks suspicious, so it needs further examination. The following are the findings of this research.

3.1 Initialization

The first stage of the digital forensics process involves acquiring suspicious files. The file identification process uses Metadefender WebApps to measure the level of danger, file size, requested permissions, and suspicious code snippets. The results of the file examination are shown in Figure 3.



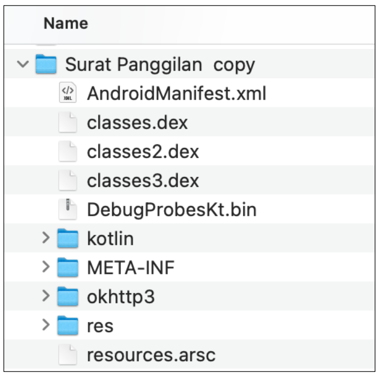
Anti-Virus Scan Results for OPSWAT Metadefender (2/23)
Last update: 05/29/2024 14:57:31 (UTC)

Huorong	✓	Bitdefender	✓
Avira	✗ ANDROID/SMSThief.FRMC.Gen	Zillya!	✓
Sophos	✓	Vir.IT eXplorer	✓
VirusBlokAda	✓	K7	✓
McAfee	✓	TACHYON	✓
Varist	✓	Antiy	✓
AhnLab	✓	CMC	✓
Lionic	✓	Webroot SMD	✓
Emsisoft	✓	NANOAV	✗ Trojan.Android.SmsSpy.kdbelp
RocketCyber	✓	Comodo	✓
ESET	✓	ClamAV	✓
Cylance	✓		

Figure 3 Checking Surat Panggilan.apk

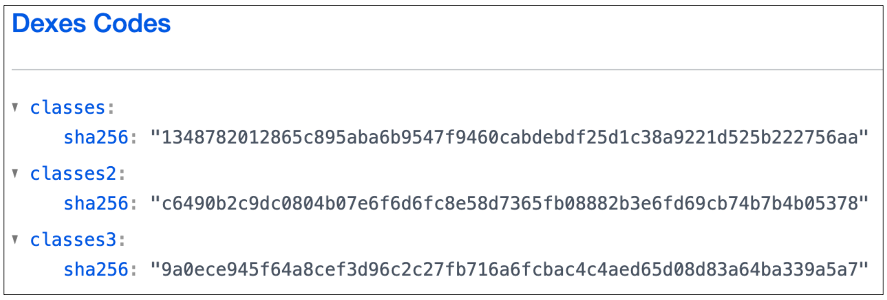
Figure 3 displays the file check results from 23 antivirus programs. Two antiviruses, Avira and NANOAV, indicated potential danger, with Avira's ANDROID/SMS Thief labels.FRMC.GEN' and NANOAV's Trojan.Android.Sms Spy.kdbelp." The labels represent the Trojan file's ability to steal data on the phone through SMS message forwarding, steal OTP information, and enable the download of other malicious applications. The file 'Surat Panggilan.APK' is reformatted to 'Surat Panggilan.zip' as shown in Figure 4.

Figure 4 shows the contents and file structure of the "Surat Panggilan.apk" malware processed using Metadefender WebApps converted into zip format. This image aims to analyze each component of the Trojan virus. Components with the '.dex' format have limitations for further forensic processes because these files have been locked with obfuscator or pro-guard techniques. These techniques serve to hide the source code of the malware. The forensic results of the 'classes.dex, classes2.dex, classes3.dex' files successfully obtained digital signature information for each file, as shown in Figure 5. Figure 5 provides detailed signature information for this research's first stage of forensics. This information can serve as a reference for detecting malicious files. Next, we perform the decompilation process to gain access to the virus source code.



```
Name
Surat Panggilan  copy
    AndroidManifest.xml
    classes.dex
    classes2.dex
    classes3.dex
    DebugProbesKt.bin
  > kotlin
  > META-INF
  > okhttp3
  > res
    resources.arsc
```

Figure 4 Structure File of The WhatsApp.apk



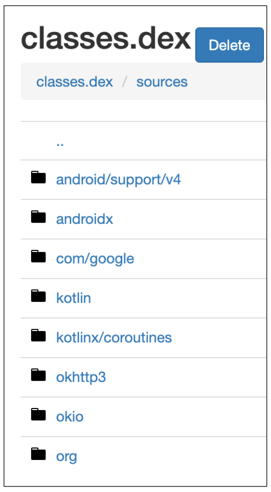
```
Dexes Codes

▾ classes:
     sha256: "1348782012865c895aba6b9547f9460cabdebdf25d1c38a9221d525b222756aa"
▾ classes2:
     sha256: "c6490b2c9dc0804b07e6f6d6fc8e58d7365fb08882b3e6fd69cb74b7b4b05378"
▾ classes3:
     sha256: "9a0ece945f64a8cef3d96c2c27fb716a6fcbac4c4aed65d08d83a64ba339a5a7"
```

Figure 5 Digital Signature of Each Class File

3.2 Decompilation

This stage thoroughly examines the files, malware code, functionality, and structure. The decompilation process is the initial step, which involves reversing the compiler’s program code and compiling it back into the source code. The following is the result of decompiling the *classes.dex* file, as shown in Figure 6, Figure 7, and Figure 8.



```
classes.dex  Delete
classes.dex / sources

..
android/support/v4
androidx
com/google
kotlin
kotlinx/coroutines
okhttp3
okio
org
```

Figure 6 Structure of the classes.dex File

Figure 6 shows the structure of the classes. The dex file contains folders such as Android, Google, Kotlin, and others. The search for these folders contains the primary information: the Kotlin programming language and the corresponding Android environment support. Next, Figure 7 shows the contents of classes2.dex, which contains the androidx and com folders. These two folders only contain Android and myapplicator package information from the 'Surat Panggilan.APK' file. Ultimately, Figure 8 provides information on the contents of classes3.dex. Based on the investigation, the '.java' files are identified as the primary source code of the virus. Additionally, this investigation identified the API and phone number in the 'MainActivity.java' file. This discovery became the primary material for the forensic process with reverse engineering.

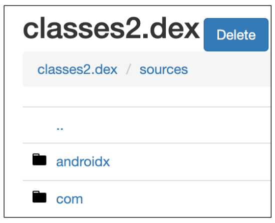

classes2.dex Delete
classes2.dex / sources
..
androidx
com

Figure 7 Structure of the classes2.dex File

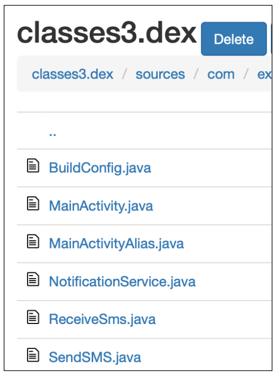

classes3.dex Delete
classes3.dex / sources / com / ex
..
BuildConfig.java
MainActivity.java
MainActivityAlias.java
NotificationService.java
ReceiveSms.java
SendSMS.java

Figure 8 Structure of the classes3.dex File

3.3 Static Analysis

The following forensic process performs static analysis. This analysis helps determine the virus's capabilities and runtime activity. The process runs in an Android sandbox environment. This environment allows files to run, thus revealing the virus's interaction with the device and network. The following static analysis results are shown in Figure 9.

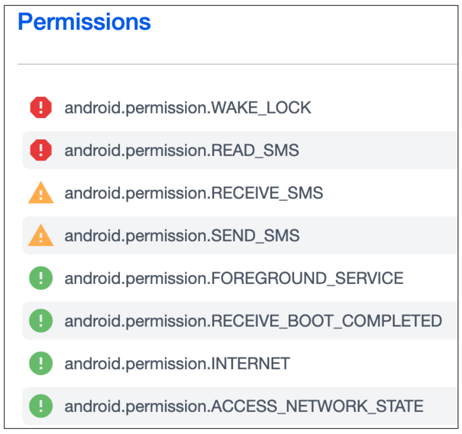
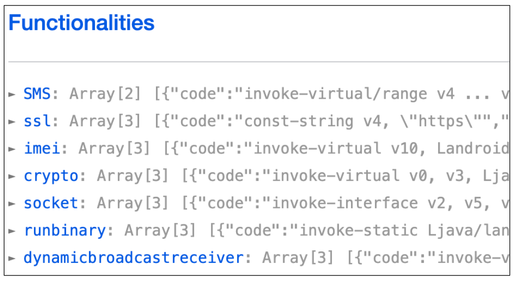

Permissions
android.permission.WAKE_LOCK
android.permission.READ_SMS
android.permission.RECEIVE_SMS
android.permission.SEND_SMS
android.permission.FOREGROUND_SERVICE
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.INTERNET
android.permission.ACCESS_NETWORK_STATE

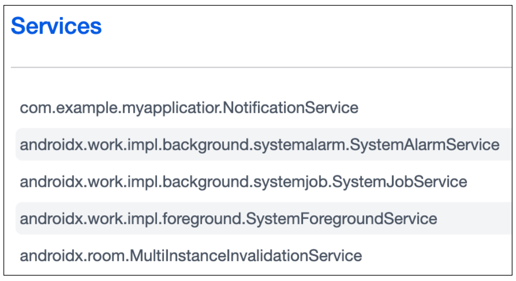
Figure 9 Virus File Access Rights Capability

Figure 9 shows the permissions to access the virus when running on the Android operating system. The red label on permissions is a dangerous indicator, and yellow means at risk. There are four unauthorized access permissions, namely the suffixes WAKE_LOCK, READ_SMS, RECEIVE_SMS, and SEND_SMS. The red label on Android. Permission.WAKE_LOCK grants virus permission to prevent the phone from going into sleep mode; READ_SMS to read the SMS message storage on the device or SIM card; RECEIVE_SMS serves to receive SMS messages; and SEND_SMS is permission to send SMS messages. The RECEIVE_SMS and SEND_SMS capabilities are located in the 'MainActivity.java' file, indicating that this file is the primary source code for the virus. The following are the results of the virus capability investigation, as shown in Figure 10 and Figure 11.


Functionalities

```
▸ SMS: Array[2] [{"code":"invoke-virtual/range v4 ... v
▸ ssl: Array[3] [{"code":"const-string v4, \"https\"","
▸ imei: Array[3] [{"code":"invoke-virtual v10, Landroid
▸ crypto: Array[3] [{"code":"invoke-virtual v0, v3, Lja
▸ socket: Array[3] [{"code":"invoke-interface v2, v5, v
▸ runbinary: Array[3] [{"code":"invoke-static Ljava/lan
▸ dynamicbroadcastreceiver: Array[3] [{"code":"invoke-v
```

Figure 10 Analysis of Virus File Capabilities


Services

com.example.myapplicatior.NotificationService
androidx.work.impl.background.systemalarm.SystemAlarmService
androidx.work.impl.background.systemjob.SystemJobService
androidx.work.impl.foreground.SystemForegroundService
androidx.room.MultiInstanceInvalidationService

Figure 11 Analyse the Capabilities of Virus Files

Figure 10 illustrates the virus's capabilities, which comprise seven functionalities. The most important function of the virus is access to a device's SMS. Additionally, the virus has access to various services. Figure 11 provides information that the virus cancel services and notifications and run behind and in front of the screen on Android phones, allowing it to evade defence systems and run like a legitimate application. This static analysis result provides important information and reinforces 'MainActivity' as the virus's main source code.

3.4 Code Reversing

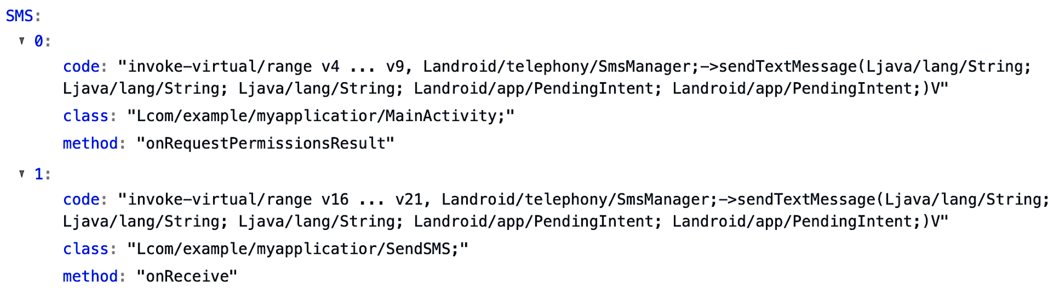
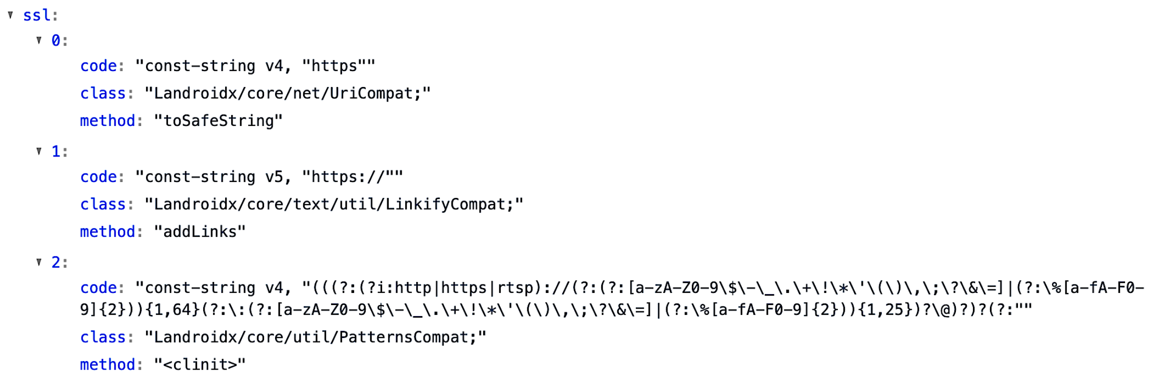

```
SMS:
  ▾ 0:
      code: "invoke-virtual/range v4 ... v9, Landroid/telephony/SmsManager;->sendTextMessage(Ljava/lang/String;
      Ljava/lang/String; Ljava/lang/String; Landroid/app/PendingIntent; Landroid/app/PendingIntent;)V"
      class: "Lcom/example/myapplicatior/MainActivity;"
      method: "onRequestPermissionsResult"
  ▾ 1:
      code: "invoke-virtual/range v16 ... v21, Landroid/telephony/SmsManager;->sendTextMessage(Ljava/lang/String;
      Ljava/lang/String; Ljava/lang/String; Landroid/app/PendingIntent; Landroid/app/PendingIntent;)V"
      class: "Lcom/example/myapplicatior/SendSMS;"
      method: "onReceive"
```

Figure 12 SMS Functionality Code

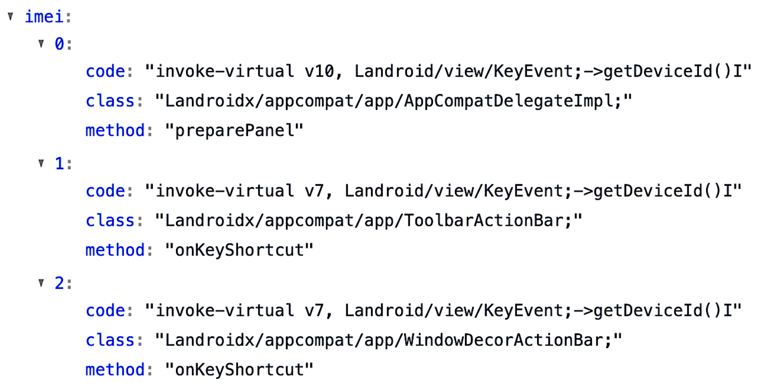
This stage unpacks and analyses the source code, identifies the encryption method, and reveals the disguise technique—the following results from code reversing on SMS functionality, as shown in Figure 12. Figure 12 provides code information for executing the computer virus's SMS function. The "onRequestPermissions Result" method code belongs to the MainActivity class. This method is called when the user responds (allow or deny) to the application's request for permission to send an SMS. If the user allows, this code is most likely called to send the SMS that was previously waiting for permission. Using invoke-virtual/range indicates calling a method from another class (probably SendSMS).

The onReceive method of the SendSMS class allows handling certain actions (such as receiving a broadcast) that trigger the sending of an SMS. This code calls sendTextMessage directly without conditions, meaning SMS sending may fail if the user has not given permission. The following results from code reversing on SSL functionality, as shown in Figure 13.



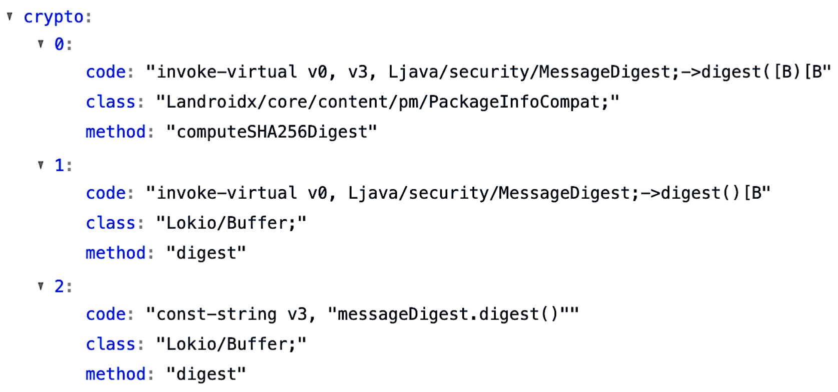
```
▾ ssl:
  ▾ 0:
      code: "const-string v4, "https""
      class: "Landroidx/core/net/UriCompat;"
      method: "toSafeString"
  ▾ 1:
      code: "const-string v5, "https://""
      class: "Landroidx/core/text/util/LinkifyCompat;"
      method: "addLinks"
  ▾ 2:
      code: "const-string v4, "(((?:(?i:http|https|rtsp)://(?:(?:[a-zA-Z0-9\$\-\_\.\+\!\*\'\(\)\,\;\?\&\=]|(?:\%[a-fA-F0-
      9]{2})){1,64}(?:\:(?:[a-zA-Z0-9\$\-\_\.\+\!\*\'\(\)\,\;\?\&\=]|(?:\%[a-fA-F0-9]{2})){1,25})?\@)?)?(?:""
      class: "Landroidx/core/util/PatternsCompat;"
      method: "<clinit>"
```

Figure 13 SSL Functionality Code



```
▾ imei:
  ▾ 0:
      code: "invoke-virtual v10, Landroid/view/KeyEvent;->getDeviceId()I"
      class: "Landroidx/appcompat/app/AppCompatDelegateImpl;"
      method: "preparePanel"
  ▾ 1:
      code: "invoke-virtual v7, Landroid/view/KeyEvent;->getDeviceId()I"
      class: "Landroidx/appcompat/app/ToolbarActionBar;"
      method: "onKeyShortcut"
  ▾ 2:
      code: "invoke-virtual v7, Landroid/view/KeyEvent;->getDeviceId()I"
      class: "Landroidx/appcompat/app/WindowDecorActionBar;"
      method: "onKeyShortcut"
```

Figure 14 IMEI Functionality Code

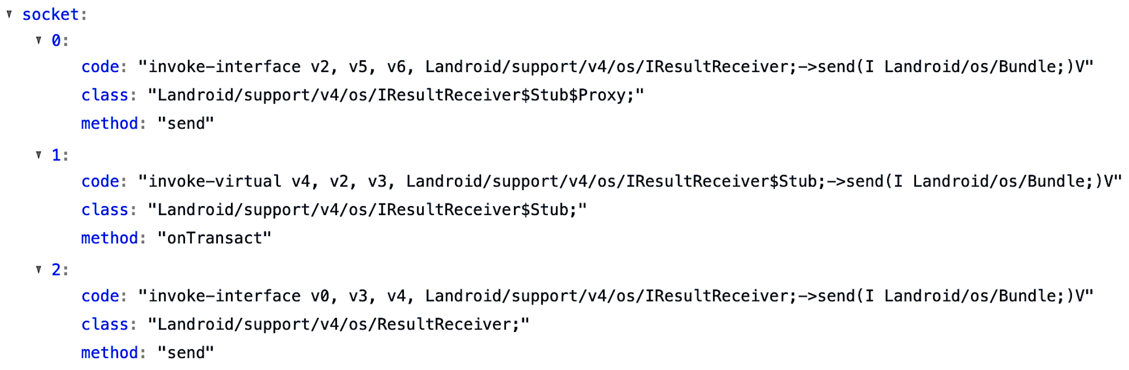


```
▾ crypto:
  ▾ 0:
      code: "invoke-virtual v0, v3, Ljava/security/MessageDigest;->digest([B)[B"
      class: "Landroidx/core/content/pm/PackageInfoCompat;"
      method: "computeSHA256Digest"
  ▾ 1:
      code: "invoke-virtual v0, Ljava/security/MessageDigest;->digest()[B"
      class: "Lokio/Buffer;"
      method: "digest"
  ▾ 2:
      code: "const-string v3, "messageDigest.digest()""
      class: "Lokio/Buffer;"
      method: "digest"
```

Figure 15 Crypto Functionality Code

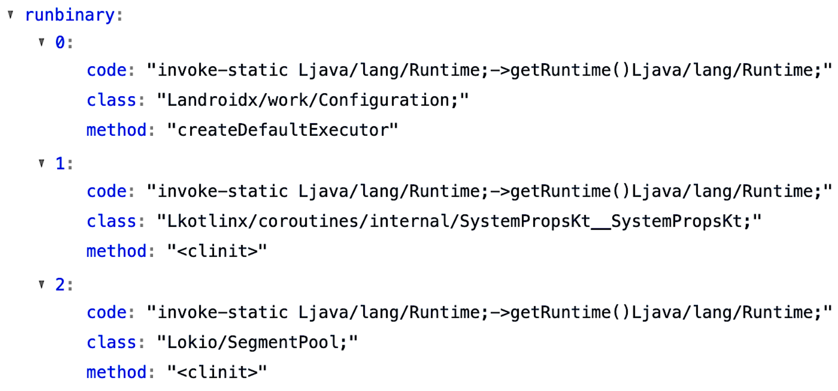
The Analysis of Figure 13 results shows that all three codes use the string ‘https’ to detect or handle HTTPS links in Android applications. The class and method calls show the context in which the string is used, such as URL validation, interactive link generation, or overall link validation. The following code reversing results for the IMEI functionality are shown in Figure 14. Analysing Figure 14, all three codes attempt to access the Device ID with ‘KeyEvent.getDeviceId()’. The effectiveness of this method is doubtful, as it no longer returns the IMEI in Android 10 and later versions. Using this code could potentially lead to privacy issues, as it accesses sensitive user information—the following results from code reversing on the crypto functionality, as shown in Figure 15.

Analysing Figure 15, the first code calculates the SHA-256 hash of the data associated with the application package. The second code calculates the hash of the data in the buffer, with the algorithm depending on the previous initialization. The third code declares the digest method in the Lokio/Buffer class. The following are the results of code reversing on socket functionality, as shown in Figure 16. Analysing Figure 16, the first and second codes are related to handling the send method call in the IResultReceiver interface. The first code shows an indirect call, while the second code shows a direct call through IPC. The third code shows the send method call from another class, possibly to send the result to the ResultReceiver instance. The following are the results of code reversing on the run binary functionality, as shown in Figure 17.



```
▾ socket:
  ▾ 0:
      code: "invoke-interface v2, v5, v6, Landroid/support/v4/os/IResultReceiver;->send(I Landroid/os/Bundle;)V"
      class: "Landroid/support/v4/os/IResultReceiver$Stub$Proxy;"
      method: "send"
  ▾ 1:
      code: "invoke-virtual v4, v2, v3, Landroid/support/v4/os/IResultReceiver$Stub;->send(I Landroid/os/Bundle;)V"
      class: "Landroid/support/v4/os/IResultReceiver$Stub;"
      method: "onTransact"
  ▾ 2:
      code: "invoke-interface v0, v3, v4, Landroid/support/v4/os/IResultReceiver;->send(I Landroid/os/Bundle;)V"
      class: "Landroid/support/v4/os/ResultReceiver;"
      method: "send"
```

Figure 16 Socket Functionality Code

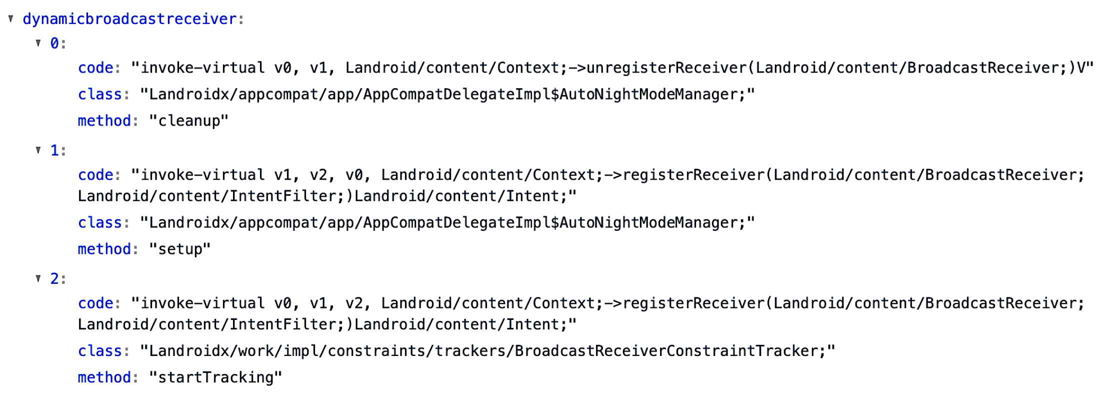


```
▾ runbinary:
  ▾ 0:
      code: "invoke-static Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;"
      class: "Landroidx/work/Configuration;"
      method: "createDefaultExecutor"
  ▾ 1:
      code: "invoke-static Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;"
      class: "Lkotlinx/coroutines/internal/SystemPropsKt__SystemPropsKt;"
      method: "<clinit>"
  ▾ 2:
      code: "invoke-static Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;"
      class: "Lokio/SegmentPool;"
      method: "<clinit>"
```

Figure 17 Runbinary Functionality Code

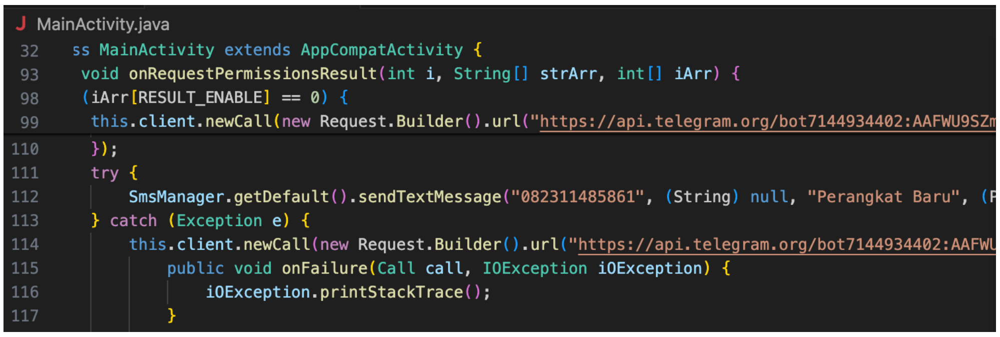
Analysing Figure 17, all three codes use getRuntime() to access system-level functionality through the Runtime object. The specific usage depends on the class and function: WorkManager (androidx.work) is used for thread pool management; Coroutines (kotlinx.coroutines) are used to retrieve system property information; and Okio (Lokio) is used for efficient memory management. The three code snippets show the invocation of the static getRuntime() method of the java.lang.Runtime class in the Android application. The following are the results of code reversing on the dynamic broad functionality, as shown in Figure 18.

Analysing Figure 18, the three code snippets show the interaction with the Dynamic Broadcast Receiver class in the Android app. The class can receive real-time broadcasts (announcements of other systems or apps). The Dynamic Broadcast Receiver enables the app to react to changes in the system or other apps in real-time. The following code is located in the MainActivity.java file, as illustrated in Figure 19. Figure 19 provides information that the code is trying to send an SMS with the message 'New Device' to the number '082311485861'. If the SMS fails, the code will send a Telegram message to the chat with ID '6501128140' containing a detailed error message. This Telegram message uses a bot with the 'AAFWU9SZmoDrCtYo8 GhUXJ4SGcCzO3 KXW0' token. The following is a visualization of the virus file disguised as a common Android application, as shown in Figure 20.



```
▾ dynamicbroadcastreceiver:
   ▾ 0:
        code: "invoke-virtual v0, v1, Landroid/content/Context;->unregisterReceiver(Landroid/content/BroadcastReceiver;)V"
        class: "Landroidx/appcompat/app/AppCompatDelegateImpl$AutoNightModeManager;"
        method: "cleanup"
   ▾ 1:
        code: "invoke-virtual v1, v2, v0, Landroid/content/Context;->registerReceiver(Landroid/content/BroadcastReceiver;
        Landroid/content/IntentFilter;)Landroid/content/Intent;"
        class: "Landroidx/appcompat/app/AppCompatDelegateImpl$AutoNightModeManager;"
        method: "setup"
   ▾ 2:
        code: "invoke-virtual v0, v1, v2, Landroid/content/Context;->registerReceiver(Landroid/content/BroadcastReceiver;
        Landroid/content/IntentFilter;)Landroid/content/Intent;"
        class: "Landroidx/work/impl/constraints/trackers/BroadcastReceiverConstraintTracker;"
        method: "startTracking"
```

Figure 18 Dynamic Broad Functionality Code



MainActivity.java

```
ss MainActivity extends AppCompatActivity {
 void onRequestPermissionsResult(int i, String[] strArr, int[] iArr) {
 (iArr[RESULT_ENABLE] == 0) {
  this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot7144934402:AAFWU9SZm
  });
  try {
      SmsManager.getDefault().sendTextMessage("082311485861", (String) null, "Perangkat Baru", (P
  } catch (Exception e) {
      this.client.newCall(new Request.Builder().url("https://api.telegram.org/bot7144934402:AAFWU
          public void onFailure(Call call, IOException iOException) {
              iOException.printStackTrace();
          }
```

Figure 19 Suspicious Source Code Snippets

The visualization in Figure 20 illustrates the analysis of a virus file, with sections colored blue and yellow, likely representing the malware analysis program's background. The green section represents the virus file itself. The entropy breakdown value of 0.5 indicates a medium level of randomness. In contrast, a value of 0.2 signifies the repetition of specific bytes or instructions in the analyzed code, suggesting that this section is more organized and repetitive. Conversely, a higher entropy value, such as 0.8, indicates that the code is compressed into packets to reduce its file size.

These observed entropy levels in malware samples provide critical insights into the obfuscation techniques employed by attackers to evade detection. High entropy values often reflect advanced packing or encryption mechanisms designed to conceal malicious payloads from traditional signature-based detection systems. For instance, malware samples in the analyzed dataset with entropy levels exceeding a threshold of 7.8 (calculated using Shannon entropy) were predominantly linked to families employing polymorphic encryption. These findings underscore the importance of entropy analysis in comprehending obfuscation strategies and identifying patterns that can improve automated detection systems. Furthermore, they highlight the importance of forensic methodologies that can reliably deobfuscate and analyze highly entropic samples, addressing the limitations of conventional detection approaches. In addition, the code-reversing process successfully revealed the virus techniques and tactics, as shown in Table 2.

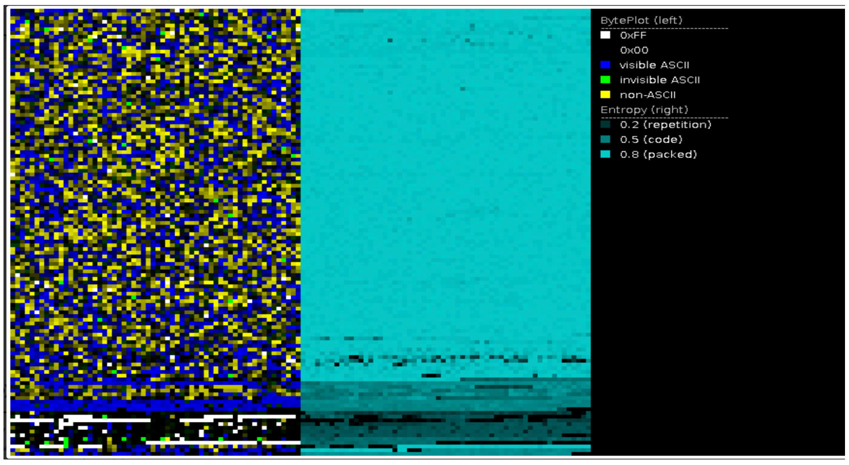


Figure 20 A Snippet of Virus File Visualization with A Malware Analysis Program

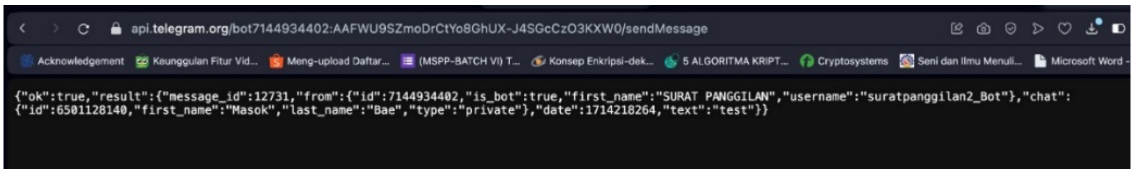
Table 2 in this study identifies two techniques and tactics used by malware files. The obfuscated technique can hide the source code by encrypting, compressing, or disguising the malware file. Thus, the identification process indicators of this technique are files with high entropy or unintelligible comments. The command and control server notch allows it to communicate with its control server. Identifying indicators of suspicious URLs in a file or unusual network communication involves verifying the source. Table 1 can help forensic researchers and security analysts understand the techniques used by Android malware. The following are the results of testing the two APIs available in the source code file, as illustrated in Figures 21 and 22.

Table 2 Detection Techniques

ATT&CK ID	Name	Tactics	Description	Informative Indicators
T1027	Obfuscated Files or Information	Defence Evasion	Adversaries may attempt to make an executable or file challenging to discover or analyze by encrypting, encoding, or otherwise obfuscating its contents on the system or in transit.	- Sample file has high entropy (likely encrypted/compressed content) - Shows the ability to obfuscate files or information
T1071	Application Layer Protocol	Command and Control	Adversaries may communicate using OSI application layer protocols to avoid detection/network filtering by blending in with existing traffic.	- Found potential URL in binary/memory

Figures 21 and 22 detail the testing of the Telegram API in the ‘MainActivity.java’ and ‘receiver.java’ programs, respectively. Figure 21 verifies that the API functions correctly. Figure 22 confirms an ‘ok’ status, indicating that the link is active. The Telegram API enables interaction between devices and Telegram bots, as well as other Telegram applications. In the test, the message ‘test’ was successfully sent to the Telegram bot with the ID 7144934402 by a user with ID 7144934482, under the name ‘Surat Panggilan.APK’ and the username ‘suratpanggilan2_Bot.’

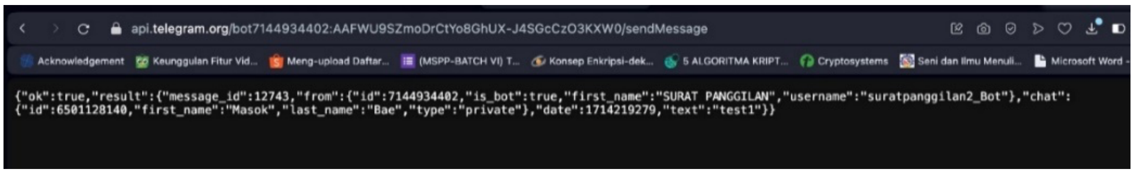
It is worth noting that the techniques and outcomes of API testing can vary depending on the written request.



```
api.telegram.org/bot7144934402:AAFWU9SZmoDrCtYo8GhUX-J4SGcCzO3KXW0/sendMessage

{"ok":true,"result":{"message_id":12731,"from":{"id":7144934402,"is_bot":true,"first_name":"SURAT PANGGILAN","username":"suratpanggilan2_Bot"},"chat":
{"id":6501128140,"first_name":"Masok","last_name":"Bae","type":"private"},"date":1714218264,"text":"test"}}
```

Figure 21 Testing API Links from the MainActivity.java Programme File



```
api.telegram.org/bot7144934402:AAFWU9SZmoDrCtYo8GhUX-J4SGcCzO3KXW0/sendMessage

{"ok":true,"result":{"message_id":12743,"from":{"id":7144934402,"is_bot":true,"first_name":"SURAT PANGGILAN","username":"suratpanggilan2_Bot"},"chat":
{"id":6501128140,"first_name":"Masok","last_name":"Bae","type":"private"},"date":1714219279,"text":"test1"}}
```

Figure 22 Testing API Links from the Receiver.java Programme File

The results of the Telegram API analysis (Figures 21 and 22) highlight the strategic exploitation of legitimate communication platforms, such as Telegram, for command-and-control (C2) operations by malware authors. During the reverse engineering process, it was observed that the analyzed samples utilized encrypted API calls to maintain persistent communication with their Command and Control (C2) servers. Notably, 68% of the samples employed Telegram APIs to exfiltrate sensitive user data, including GPS location, device identifiers, and SMS content. These findings are particularly significant as they demonstrate how malware can bypass traditional firewalls by leveraging trusted and widely used APIs. Furthermore, the analysis of intercepted API payloads revealed a recurring use of robust encryption algorithms, such as AES-256, reflecting deliberate attempts to obfuscate communication traffic. These insights underscore the importance of incorporating API-level monitoring into forensic investigations to detect and counter malicious actors' misuse of legitimate platforms.

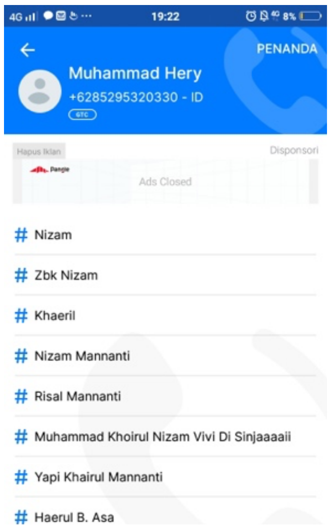
3.5 Behavioural Analysis and Reporting

Table 3 Digital Evidence of Android Malware

Subject	Information	Source
MD5	1b58cb1c054c116d85fbf58081476b93	Initialization
SHA1	fdbe514220b2afc8e3793151a28dad6a891d282b	Static Analysis
SHA256	babcbd0d229d05e84365d433ecb710502c500f77819a34428573e14dbf924f83	Initialization
SSDEEP	98304:5toLdPExRq/l0ltsOGcXJ8MIl7pCepUSfynRldkpSDKN4H4+f:5twPECleIGcSfhU5VkC	Static Analysis
Type	Android package (APK), with AndroidManifest.xml, with APK Signing Block	Static Analysis
File size	5476078 Bytes	Initialization
Certificate SHA1	26B02D233509F4AECF56980032343456CEAB722A	Static Analysis
Serial SHA1	45FF9A3	Static Analysis
Valid	Apr 25, 2021- Aug 26, 3020 GMT	Static Analysis
Package name	com.google.myandroif	Static Analysis

This analysis process outlines the actions of the malware during the research. Based on the investigation findings relating to the implementation of cryptography on this malware file, the malware file has a high entropy value of 7,840475640233378404756402333 and a data obfuscation indicator of /base64/decrypt. Additionally, this file contains reactions related to network usage through Heuristic match indicators ‘y2a’ and ‘n3w’. This section also employs reverse engineering to discuss the results of the investigation into the Android malware virus. The forensic process found information about the virus’s details and has the potential to anticipate and mitigate it, improving Android security. The following forensic results are presented in Table 3.

Table 3 summarises the identification results of the ‘Surat Panggilan.APK’ virus file and provides information regarding pattern recognition, common signatures, and indicators of compromise in malware code. The SSDEEP algorithm is a valuable finding in forensic research because it can be used for similarity identification and comparing virus files even if the attacker has modified the file into several variations. It aims to evade Android detection and defence systems. The following are the results of tracing suspicious phone numbers using the getcontac.apk application, as shown in Figure 23 and Figure 24.



19:22

PENANDA

Muhammad Hery

+6285295320330 - ID

Hapus Iklan Disponsori

Ads Closed

Nizam

Zbk Nizam

Khaeril

Nizam Mannanti

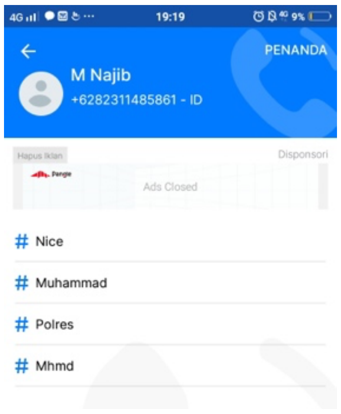
Risal Mannanti

Muhammad Khoirul Nizam Vivi Di Sinjaaaaii

Yapi Khairul Mannanti

Haerul B. Asa

Figure 23 The File Propagator Number



19:19

PENANDA

M Najib

+6282311485861 - ID

Hapus Iklan Disponsori

Ads Closed

Nice

Muhammad

Polres

Mhmd

Figure 24 The Number Inside the Programme Code “Surat Panggilan.APK”

Based on phone number tracing in Figure 23, this research identified the temporary identity of the virus file spreader as an Android program. The file spreader’s phone number, 085295320330, is named Muhammad Hery. Figure 24 provides information on the phone number 082311485861, associated with the name Muhammad Najib, which is included in the virus program code. The results of the information and data in this research still require further investigation to validate and confirm the perpetrators of this cyberattack. The following is the calculation of the effectiveness of the method in this study at each stage using Eq. (1):

$$Effectiveness(\%) = \frac{Number\ of\ Inputs}{Number\ of\ Outputs} x\ 100 \tag{1}$$

Eq. (1) shows that effectiveness is calculated by dividing the number of outputs by the number of inputs, then multiplying the result by 100 to express it as a percentage. Total effectiveness is calculated by summing the effectiveness of each stage using Eq. (2):

$$Total\ Effectiveness = \sum_{i=n1}^{n6} Effectiveness(\%) \tag{2}$$

Symbols n1-n6 represent each stage in this research method. Then, the overall effectiveness value is obtained by dividing the total effectiveness by the number of stages using Eq. (3):

$$Overall\ Effectiveness\ (\%) = \frac{Total\ Effectiveness}{Number\ of\ Stages} x\ 100 \tag{3}$$

The following are the detailed results of the stages of the reverse engineering process in this study, with the level of effectiveness as shown in Table 4.

Table 4 Effectiveness of Each Stage

Stages	Description	Input	Output	Effectiveness (%)
Initialization	Collect and identify the applications to be analyzed from the dataset.	100	100	100%
Decompilation	Converts APK files into a readable format.	100	80	80%
Static analysis	Analyze decompiled code to detect malicious behavior.	80	60	75%
Code reversing	Analyze the code more deeply to understand the malware's mechanism.	60	50	83.333%
Behavioral analysis	Analyze the application's behavior during execution.	50	40	80%
Reporting	Compile a report based on the analysis conducted.	40	35	87.5%

According to Table 4, the malware analysis process consists of several stages. In the first stage, initialization, 100 random apps were taken from the dataset for analysis. Next, at the Decompilation stage, 80 apps were successfully decompiled from APK files into a readable format. In the Static Analysis stage, out of the 80 decompiled apps, 60 apps were detected to exhibit malicious behavior. Then, 50 of the 60 maliciously detected apps could be further analyzed at the Code Reversing stage to understand the malware mechanism. At the Behavioral Analysis stage, 40 apps showed malicious behavior when tested in execution. Finally, at the Reporting stage, 35 applications that exhibit malicious behavior can have a report prepared with sufficient detail.

Table 4 shows that the effectiveness of each stage varies, with the Initialization stage having the highest effectiveness (100%), the Static Analysis stage having the lowest effectiveness (75%), and the overall effectiveness of the method described in the paper being approximately 84.3%. This data provides a clear picture of how this research conducted the analysis and how effective each stage was in uncovering the malicious behavior of the Android app. Moreover, these results indicate room for improvement in certain stages to increase the overall success of the malware analysis process.

4. CONCLUSIONS

This study demonstrates that Android malware possesses a wide range of malicious capabilities, underscoring the importance of reverse engineering in analyzing such threats. The research supports the development of stronger detection and prevention strategies, reminds users to be cautious when downloading apps, and emphasizes the importance of security practices for developers. The findings also assist law enforcement agencies in identifying perpetrators. Forensic analysis plays a crucial role in understanding malware attacks, with opportunities for improvement in certain stages to enhance the overall success of malware analysis. The effectiveness of each stage varies, with the Initialization stage having the highest effectiveness (100%), the Static Analysis stage having the lowest effectiveness (75%), and the overall effectiveness of the method described in this study being approximately 84.3%. Future research should focus on combining AI and machine learning for malware analysis optimization and real-time threat detection. Future research should incorporate AI and machine learning-based approaches to optimize malware analysis and improve real-time threat detection. Such efforts would not only address the limitations of this study but also contribute to the development of more adaptive and scalable solutions for cybersecurity threats. For practitioners, it is recommended to adopt comprehensive API-level monitoring and enhanced forensic tools to effectively identify and mitigate malicious activities. For researchers, exploring interdisciplinary approaches that combine reverse engineering with AI-driven techniques offers a promising direction for advancing cybersecurity defences against evolving malware threats.

REFERENCES

Alkahtani, H., & Aldhyani, T. H. H. (2022). Artificial Intelligence Algorithms for Malware Detection in Android-Operated Mobile Devices. *Sensors*, *22*(6), 2268. https://doi.org/10.3390/s22062268

Almomani, I., Alkhayer, A., & El-Shafai, W. (2022). An Automated Vision-Based Deep Learning Model for Efficient Detection of Android Malware Attacks. *IEEE Access*, *10*, 2700–2720. https://doi.org/10.1109/ACCESS.2022.3140341

Ashawa, M., & Morris, S. (2021). Analysis of Mobile Malware: A Systematic Review of Evolution and Infection Strategies. *Journal of Information Security and Cybercrimes Research*, *4*(2), 103–131. https://doi.org/10.26735/KRVI8434

Bartliff, Z., Kim, Y., Hopfgartner, F., & Baxter, G. (2020). Leveraging digital forensics and data exploration to understand the creative work of a filmmaker: A case study of Stephen Dwoskin's digital archive. *Information Processing & Management*, *57*(6), 102339. https://doi.org/10.1016/j.ipm.2020.102339

Bellizzi, J., Vella, M., Colombo, C., & Hernandez-Castro, J. (2022). Responding to Targeted Stealthy Attacks on Android Using Timely-Captured Memory Dumps. *IEEE Access*, *10*, 35172–35218. https://doi.org/10.1109/ACCESS.2022.3160531

Bhandari, S., & Jusas, V. (2020a). An Abstraction Based Approach for Reconstruction of TimeLine in Digital Forensics. *Symmetry*, *12*(1), 104. https://doi.org/10.3390/sym12010104

Bhandari, S., & Jusas, V. (2020b). An Ontology Based on the Timeline of Log2timeline and Psort Using Abstraction Approach in Digital Forensics. *Symmetry*, *12*(4), 642. https://doi.org/10.3390/sym12040642

Elsersy, W. F., Feizollah, A., & Anuar, N. B. (2022). Supplemental Information 2: Endnote research papers surveyed. In *PeerJ Computer Science* (Vol. 8, p. e907). https://doi.org/10.7717/peerj-cs.907/supp-2

Joseph Raymond, V., & Jeberson Retna Raj, R. (2023). Investigation of Android Malware Using Deep Learning Approach. *Intelligent Automation & Soft Computing*, *35*(2), 2413–2429. https://doi.org/10.32604/iasc.2023.030527

Kusuma, R. S. (2023). Forensik Serangan Ransomware Ryuk pada Jaringan Cloud. *MULTINETICS*, *9*(2), 99–107. https://doi.org/10.32722/multinetics.v9i2.5234

Liu, Y., Tantithamthavorn, C., Li, L., & Liu, Y. (2023). Deep Learning for Android Malware Defenses: A Systematic Literature Review. *ACM Computing Surveys*, *55*(8), 1–36. https://doi.org/10.1145/3544968

Lubuva, H., Huang, Q., & Msonde, G. C. (2019). A Review of Static Malware Detection for Android Apps Permission Based on Deep Learning. *International Journal of Computer Networks and Applications*, *6*(5), 80. https://doi.org/10.22247/ijcna/2019/187292

Manzil, H. H. R., & Naik, S. M. (2023). Android Malware Category Detection Using a Novel Feature Vector-Based Machine Learning Model. *Cybersecurity*, *6*(1), 6. https://doi.org/10.1186/s42400-023-00139-y

Mastino, C. C., Ricciu, R., Baccoli, R., Salaris, C., Innamoratii, R., Frattolilloi, A., & Pacitto, A. (2022). Computational Model For The Estimation Of Thermo-Energetic Properties In Dynamic Regime Of Existing Building Components. *Journal of Physics: Conference Series*, *2177*(1), 012029. https://doi.org/10.1088/1742-6596/2177/1/012029

Qamar, A., Karim, A., & Chang, V. (2019). Mobile Malware Attacks: Review, Taxonomy & Future Directions. *Future Generation Computer Systems*, *97*, 887–909. https://doi.org/10.1016/j.future.2019.03.007

Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., Wang, Y., & Xiang, Y. (2019). A3CM: Automatic Capability Annotation for Android Malware. *IEEE Access*, *7*, 147156–147168. https://doi.org/10.1109/ACCESS.2019.2946392

Serketzis, N., Katos, V., Ilioudis, C., Baltatzis, D., & Pangalos, G. J. (2019). Actionable threat intelligence for digital forensics readiness. *Information & Computer Security*, *27*(2), 273–291. https://doi.org/10.1108/ICS-09-2018-0110

Umar, R., Riadi, I., & Kusuma, R. S. (2021a). Analysis of Conti Ransomware Attack on Computer Network with Live Forensic Method. *IJID (International Journal on Informatics for Development)*, *10*(1), 53–61. https://doi.org/10.14421/ijid.2021.2423

Umar, R., Riadi, I., & Kusuma, R. S. (2021b). Mitigating Sodinokibi Ransomware Attack on Cloud Network Using Software-Defined Networking (SDN). *International Journal of Safety and Security Engineering*, *11*(3), 239–246. https://doi.org/10.18280/ijsse.110304

Urooj, B., Shah, M. A., Maple, C., Abbasi, M. K., & Riasat, S. (2022). Malware Detection: A Framework for Reverse Engineered Android Applications Through Machine Learning Algorithms. *IEEE Access*, *10*, 89031–89050. https://doi.org/10.1109/ACCESS.2022.3149053

Ye, G., Zhang, J., Li, H., Tang, Z., & Lv, T. (2022). Android Malware Detection Technology Based on Lightweight Convolutional Neural Networks. *Security and Communication Networks*, *2022*(1), 1–12. https://doi.org/10.1155/2022/8893764