Hanoi University of Science and Technology

School of Information and Communication Technology

Graduation Research 2
Report

Nguyen Tieu Phuong

Student ID: 20210692

Supervisor: Prof. Tran Quang Duc, Prof. Truong Thi Dieu Linh

A report submitted in partial fulfilment of the requirements of the course Graduation Research 2 as part of the *ICT Programme*

December 27, 2024

Declaration

I, Nguyen Tieu Phuong, of the School of Information and Communication Technology, Hanoi University of Science and Technology, confirm that this is my own work - figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalized accordingly.

Nguyen Tieu Phuong
December 27, 2024

Chapter 1

Introduction

1.1 Background

The increasing prevalence of Android devices has made them a major target for malware developers. Analyzing and mitigating these threats requires effective tools and methodologies to understand malicious behavior.

1.2 Problem Statement

A significant challenge faced by malware analysts today is the lack of tools capable of performing in-depth and precise analysis of suspicious files. While existing tools may flag files as malicious, they often fall short in identifying the specific code segments responsible for harmful behavior. Currently, this process is carried out manually, which is both time-consuming and error-prone. This research seeks to explore the development of tools that could assist researchers and analysts by automating and streamlining this process.

1.3 Aims and Objectives

The aim of this research is to investigate the potential of graph-based analysis in identifying code segments responsible for malicious behavior in Android malware. By leveraging existing tools and datasets, this study seeks to provide initial insights into how this approach can enhance malware analysis.

The objectives of this project are to generate graphs from APK files and gain preliminary insights into the malicious behaviors embedded within these files, contributing to more efficient and accurate malware detection.

1.4 Organization of the Report

This report is structured into five chapters. Chapter 2 provides a comprehensive overview of malware analysis and the Android architecture. Chapter 3 details the methodology used to collect datasets and explore the available tools. The results of the research are presented and discussed in Chapter 4, followed by the final conclusions and reflections in Chapter 5.

Chapter 2

Foundations

This chapter outlines malware analysis basics, Android architecture, Android components and lifecycles, and key challenges in malware analysis. It also reviews existing methods and tools, emphasizing the need for precise detection of malicious code.

2.1 What is Malware and Malware Analysis?

Malware refers to any software designed to harm or exploit users, computers, or networks. It can take various forms, such as viruses, Trojan horses, worms, rootkits, scareware, and spyware, among others.

Malware analysis aims to gather the necessary information for effective incident response. Through this process, analysts seek to understand the capabilities and behavior of malicious files, identify patterns for detection within a network, and devise strategies to mitigate, contain, and eliminate the threat.

2.2 Malware analysis approaches

Although malicious software varies widely, malware analysis techniques are generally categorized into two main approaches:

- **Static Analysis.** This method involves examining malware without executing it. Static analysis focuses on dissecting the executable file to determine its malicious intent, functionality, and potential signatures. Advanced techniques include reverse-engineering the file's internals using a disassembler to analyze program instructions in detail.
- **Dynamic Analysis.** This approach requires running the malware to observe its behavior. Dynamic analysis typically involves executing the binary within a controlled virtual environment to monitor its actions. Advanced techniques may utilize a debugger to inspect the internal state of the running executable, providing critical insights when other methods fail to yield sufficient information.

2.3 Android and Its Architecture

Understanding Android architecture is crucial for effective malware analysis because it provides insights into how applications interact with system components, such as the Linux Kernel, runtime environment, and platform libraries.

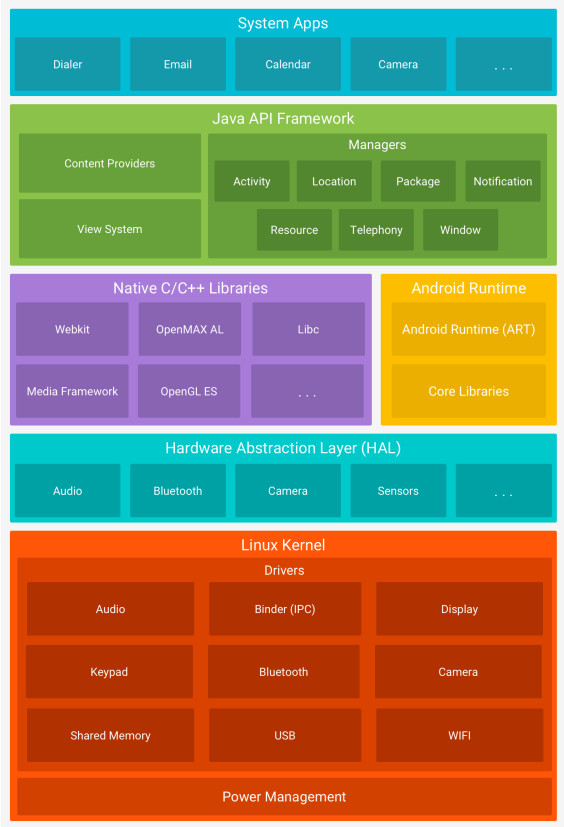


Figure 2.1: The Android software stack.

The Android architecture is composed of five main layers:

1. **Applications**
   The top layer of the Android architecture is the Applications layer, where all pre-installed and user-installed applications reside. This includes default apps like Home, Contacts, Camera, and Gallery, as well as third-party apps downloaded from the Play Store, such as chat apps and games. These applications run within the Android Runtime and utilize the classes and services provided by the underlying Application Framework layer.

2. **Application Framework**
   The Application Framework provides a comprehensive set of classes and services to support the creation of Android applications. It abstracts hardware access and simplifies the management of user interfaces and application resources. Key components include the Activity Manager, Notification Manager, View System, and Package Manager. These services allow developers to efficiently create and manage applications based on their specific requirements.

3. **Android Runtime (ART)**
   The Android Runtime (ART) is a core component of the Android architecture that powers applications through core libraries. ART compiles application bytecode into native code using Ahead-of-Time (AOT) compilation, introduced in Android 5.0 and later versions.

   Previously, Android used the Dalvik Virtual Machine (DVM), a register-based virtual machine optimized for mobile devices. ART enhances performance and memory effi-

ciency, enabling multiple instances to run simultaneously. The runtime layer relies on the Linux Kernel for threading and low-level memory management and supports development using standard Java or Kotlin programming languages.

4. **Platform Libraries**
   The Platform Libraries consist of essential C/C++ core libraries and Java-based libraries that enable robust Android development. Examples include:

   - Media Library: Supports audio and video playback and recording.
   - Surface Manager: Handles access to the display subsystem.
   - OpenGL and SGL: Provide APIs for 2D and 3D graphics.
   - SQLite: Offers lightweight database functionality.
   - FreeType: Provides font rendering.
   - WebKit: Enables web content display and page loading.
   - SSL (Secure Sockets Layer): Ensures encrypted communication for secure data transfer.

5. **Linux Kernel**
   At the core of the Android architecture is the Linux Kernel, which provides essential services such as memory management, process management, power management, and hardware abstraction. It supports device drivers for components like displays, cameras, Bluetooth, and audio. Key features of the Linux Kernel include:

   - Security: Manages secure interactions between applications and the system.
   - Memory Management: Efficiently allocates and manages memory resources.
   - Process Management: Oversees processes and resource allocation.
   - Network Stack: Facilitates seamless network communication.
   - Driver Model: Integrates device-specific drivers for optimal hardware functionality.

2.4 APK and Its Structure

An **APK (Android Package)** is the file format used to distribute and install applications on Android devices. It is essentially a compressed archive that contains all the components needed to run an Android app. APK files are similar to `.zip` files and can be opened or unpacked using standard archive tools.

The structure of an APK includes the following key components:

1. **Manifest File (`AndroidManifest.xml`)**
   This file contains essential metadata about the application, such as its package name, permissions, components (activities, services, etc.), and hardware/software requirements.

2. **DEX Files (`classes.dex`)**
   These files hold the compiled bytecode for the app, which the Android Runtime (ART) or Dalvik Virtual Machine (DVM) executes.

3. **Resources (`res/`)**
   This directory contains the app's static resources, such as images, layouts, strings, and styles used in the application interface.

4. **Resources File (`resources.arsc`)**
   This file compiles all the app's resources into a binary format for efficient access during runtime.

5. **Native Libraries (`lib/`)**
   This directory includes compiled native code (usually in C or C++) specific to the app's hardware architecture, such as ARM or x86.

6. **Assets (`assets/`)**
   The `assets/` folder contains raw, uncompiled files such as custom fonts, configuration files, or other data the app might need at runtime.

7. **META-INF Directory**
   This directory includes metadata about the APK, such as the app's certificate, signature, and manifest information required for integrity verification and secure installation.

8. **Other Files**
   Additional files may include `kotlin/` (Kotlin-specific files) or other directories related to specific tools and libraries used during development.

Understanding the structure of an APK is vital for malware analysis, as it enables analysts to inspect its components, reverse-engineer code, and identify potential malicious behaviors or hidden payloads.

2.5 Android Components and Lifecycle

Android applications are built using four primary components, each designed to perform a specific role within the application. Understanding these components and their lifecycle is essential for effective app development and analysis.

2.5.1 Activity

An **Activity** represents a single screen with a user interface in an Android app. It acts as the entry point for user interaction with the application.

Activities go through a sequence of lifecycle states:

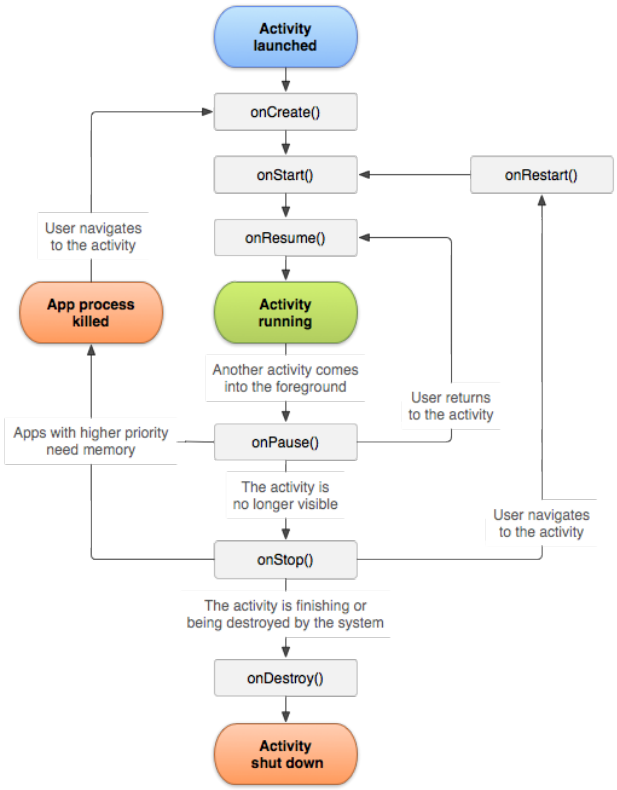


Figure 2.2: The activity lifecycle.

- `onCreate()`: Called when the activity is first created. Used to initialize the UI and essential resources.
- `onStart()`: The activity becomes visible but not interactive.
- `onResume()`: The activity enters the foreground and becomes interactive.
- `onPause()`: Called when the activity loses focus but remains partially visible (e.g., a dialog appears).
- `onStop()`: The activity is no longer visible.
- `onDestroy()`: The activity is being destroyed and resources are released.

2.5.2 Services

A **Service** performs long-running background operations without a user interface. It is useful for tasks like downloading files or playing music.

2.5.3 Broadcast Receivers

Broadcast Receivers handle communication between the Android system and the app, allowing apps to respond to system-wide broadcast events, such as low battery or connectivity changes.

2.5.4 Content Providers

Content Providers manage shared data between applications. They enable apps to access data stored in other apps or the device, such as contacts or media files.

2.6 Decompiling and Decompilers

A **compiler** is a tool that translates high-level programming languages, such as Java or Kotlin, into machine code or an intermediate representation, enabling the execution of software on a specific platform. During this process, the source code is transformed into a format that the machine understands, such as bytecode in the case of Android applications.

Decompiling, on the other hand, is the reverse process, where the compiled code is analyzed and converted back into a higher-level, human-readable format. Decompilers attempt to recover the original source code from the compiled bytecode or machine code, though this process is imperfect and may result in code that is difficult to understand.

In the context of malware analysis, decompiling is often used to examine the internals of an APK file and understand its behavior by attempting to recover the source code, which can reveal the presence of malicious intent.

2.7 Signals of Android malwares

Malware often exhibits certain telltale signs that can indicate its presence on a device. One of the most common indicators is unusual consumption of data or system resources, such as excessive battery drain, high CPU usage, or unexpected spikes in network traffic.

Another frequent sign is the sudden appearance of intrusive notifications or pop-ups, which may prompt users to install additional malicious software or provide sensitive information. Some malware may also lead to the creation of fake sign-up pages or forms designed to steal personal data, such as login credentials or financial information.

Other signs can include sluggish device performance, unexpected crashes, or the appearance of unfamiliar apps that cannot be uninstalled. Recognizing these signals is critical for identifying and mitigating the impact of malware on a system.

2.8 Summary

- Malware is harmful software, and malware analysis helps understand its behavior and mitigate damage.
- Static analysis examines malware without executing it, while dynamic analysis involves running it in a controlled environment.
- Android apps consist of four key components: Activities, Services, Content Providers, and Broadcast Receivers.
- APK files include various components, and decompiling them aids in recovering source code for analysis.
- Compilers translate code to machine language, while decompilers reverse this process to examine the source code.
- Signs of malware include excessive data usage, pop-ups, and fake sign-up pages.

Chapter 3

Methodology

Acquiring reliable datasets and identifying suitable tools are foundational steps in malware analysis. This chapter details the process of sourcing Android malware datasets, including the challenges of obtaining data from trusted sources. It also explores various tools capable of generating graphs from APK files, evaluating their functionality and relevance to the research objectives.

3.1 Data Collection

3.1.1 Collecting APK Malware Samples

The first step in this research involved collecting Android malware samples to serve as the dataset for analysis. To ensure diversity and relevance, multiple public and private sources were utilized, as described below:

- **Contagio Mini-Dump**: A well-known source for malware samples. APKs were accessed through the provided links, requiring navigation through curated malware collections.
- **Drebin Dataset**: The Drebin dataset, a benchmark for Android malware analysis, was accessed after contacting the authors via email. It provided labeled samples and metadata useful for research.
- **Canada Cyber Threat Intelligence Dataset**: This dataset from the University of New Brunswick contains a variety of Android malware APKs, accessible through their official repository.
- **VirusShare**: A large repository of malware samples. Access to the database was granted after an email request. It provided numerous APKs spanning various malware families.
- **AndroOBFS**: This dataset specializes in obfuscated Android malware. Access was facilitated through prior collaboration with the authors.

The final dataset used for this research is the MalDroid-2020 dataset (http://205.174.165.80/CICDataset/MalDroid-2020/Dataset/), which contains diverse samples of Android malware, including those with obfuscated and advanced malicious behaviors.

3.1.2 Challenges in Data Collection

During the dataset collection process, several challenges were encountered:

- **Access restrictions**: Some datasets, such as Drebin and VirusShare, required direct communication with the authors or administrators to gain access.
- **Dataset quality**: Variability in dataset quality and relevance was noted. Some sources contained incomplete or outdated samples.
- **Legal and ethical considerations**: Downloading and handling malware samples required strict adherence to ethical guidelines and safe handling practices to prevent unintentional harm.

Despite these challenges, a diverse and robust dataset was successfully collected, ensuring the comprehensiveness of this study.

The final dataset, MalDroid-2020, was chosen due to its diversity and relevance to the research objectives. The collected samples form the foundation for subsequent analysis workflows in this study.

3.2 Tools Exploration

To facilitate static analysis and prepare for the generation of graphs for Android malware samples, several tools and frameworks were explored. These tools were selected for their relevance in decompiling, reverse engineering, and analyzing APK files, as well as their ability to generate and manipulate program representations.

- **APKTool**: A powerful tool for reverse engineering Android applications. It allows for the decompilation of APKs into their respective resources and manifest files, providing insights into application structures.
- **JADX**: A decompiler used to transform APK files into readable Java source code. It aids in understanding the logic and control flow of applications.
- **Ghidra**: A comprehensive reverse engineering framework developed by the NSA. Ghidra was employed to analyze binary code, providing detailed low-level insights that complement APKTool and JADX.
- **FlowDroid**: A static analysis tool designed for precise taint analysis of Android applications. It identifies sensitive data flows, which is valuable for understanding malicious behaviors.
- **Soot Framework**: A robust Java optimization and analysis framework. Soot was utilized to analyze the control flow, generate intermediate representations, and produce graphs representing the structure and behavior of malware samples.
- **Androguard**: A powerful open-source tool for analyzing and reverse-engineering Android applications, offering features like static analysis of APKs, disassembly, and decompilation to help identify vulnerabilities, malware, and other security risks.

The primary aim of these tools was to support the comparison of malware variants within the same family. This was achieved by extracting relevant code segments and generating graphs for multiple samples to identify commonalities and differences among them. These graphs serve as the foundation for understanding the unique behaviors of malware variants and their structural characteristics.

3.3 Analysis Workflow

The analysis workflow for this research involves multiple steps to extract, analyze, and visualize the behavior and structure of Android malware samples. The process integrates various tools and techniques for static analysis, graph generation, and visualization.

- **Behavioral Analysis:** The APK file is first uploaded to websites like VirusTotal to identify its malware family and obtain a comprehensive report of its behavior.
- **Static Code Analysis:**
  - The APK file is decompiled using tools like *JADX* and *APKTool*.
  - Decompiled files include Java source code and Smali code, which are carefully studied. Special attention is given to `classes.dex` for bytecode analysis and the `AndroidManifest.xml` file to understand permissions and application configurations.
- **Graph Generation with Ghidra:**
  - The APK file is analyzed using Ghidra to generate graphs.
  - Three types of graphs are considered: code flow graphs, block flow graphs, and call graphs, each providing unique insights into the malware's structure and behavior.
- **Graph Generation with FlowDroid and Soot:**
  - FlowDroid and the Soot framework are used to perform taint analysis and generate program representations.
  - These representations are converted into visual graphs using visualization tools like *GraphViz* or *Gephi*.

The workflow ensures comprehensive static analysis, starting from high-level behavioral insights to detailed structural analysis using graph representations.

The following diagram illustrates the overall workflow:

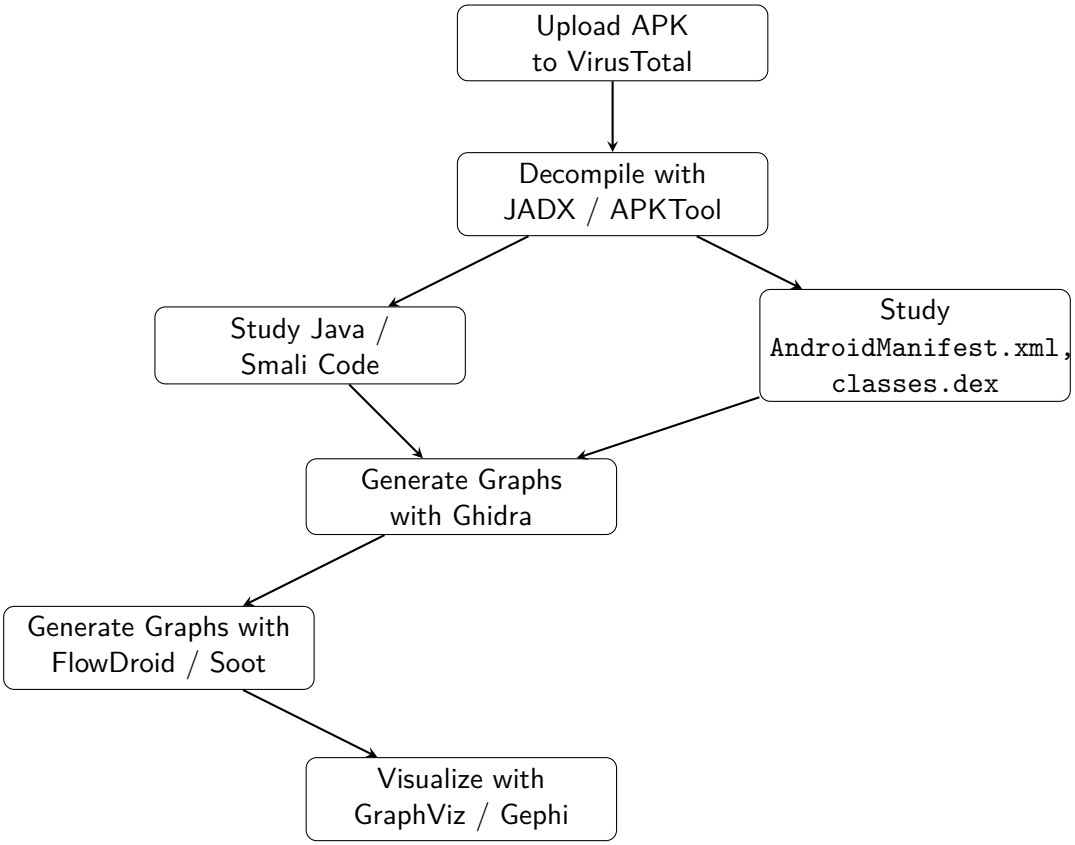


Figure 3.1: Analysis Workflow for Static Analysis of APK Files

3.4 Summary

- Malware datasets were collected from various reputable sources, ensuring a diverse range of samples for analysis.
- Key challenges in data collection included restricted access to some datasets and time delays due to permission requests.
- Tools like JADX, APKTool, Ghidra, FlowDroid, and the Soot framework were explored for static analysis and graph generation.
- The analysis workflow involved examining APK files for family identification, decompiling to study Java and Smali code, and generating various graph types for deeper insights.

Chapter 4

Results

This chapter describes the graph generation process from APK files and presents initial observations, focusing on their potential to identify malicious code.

4.1 Overview of Graph-Based Representations

Graph-based representations play a vital role in understanding the structure and behavior of malware. In this study, three primary types of graphs are utilized:

- **Call Graphs:** These graphs illustrate the relationships and interactions between different methods or functions within the code. They provide a high-level view of how components communicate.
- **Block Flow Graphs:** These graphs represent the control flow between blocks of code within a method. They help in understanding execution pathways and conditions.
- **Code Flow Graphs:** These are finer-grained graphs showing the instruction-level flow of execution. They are particularly useful for pinpointing specific malicious code segments.

Other graph types, such as data flow graphs or dependency graphs, can also aid in understanding complex malware behavior. However, this study focuses primarily on the three aforementioned graph types.

4.2 Graph Generation Process

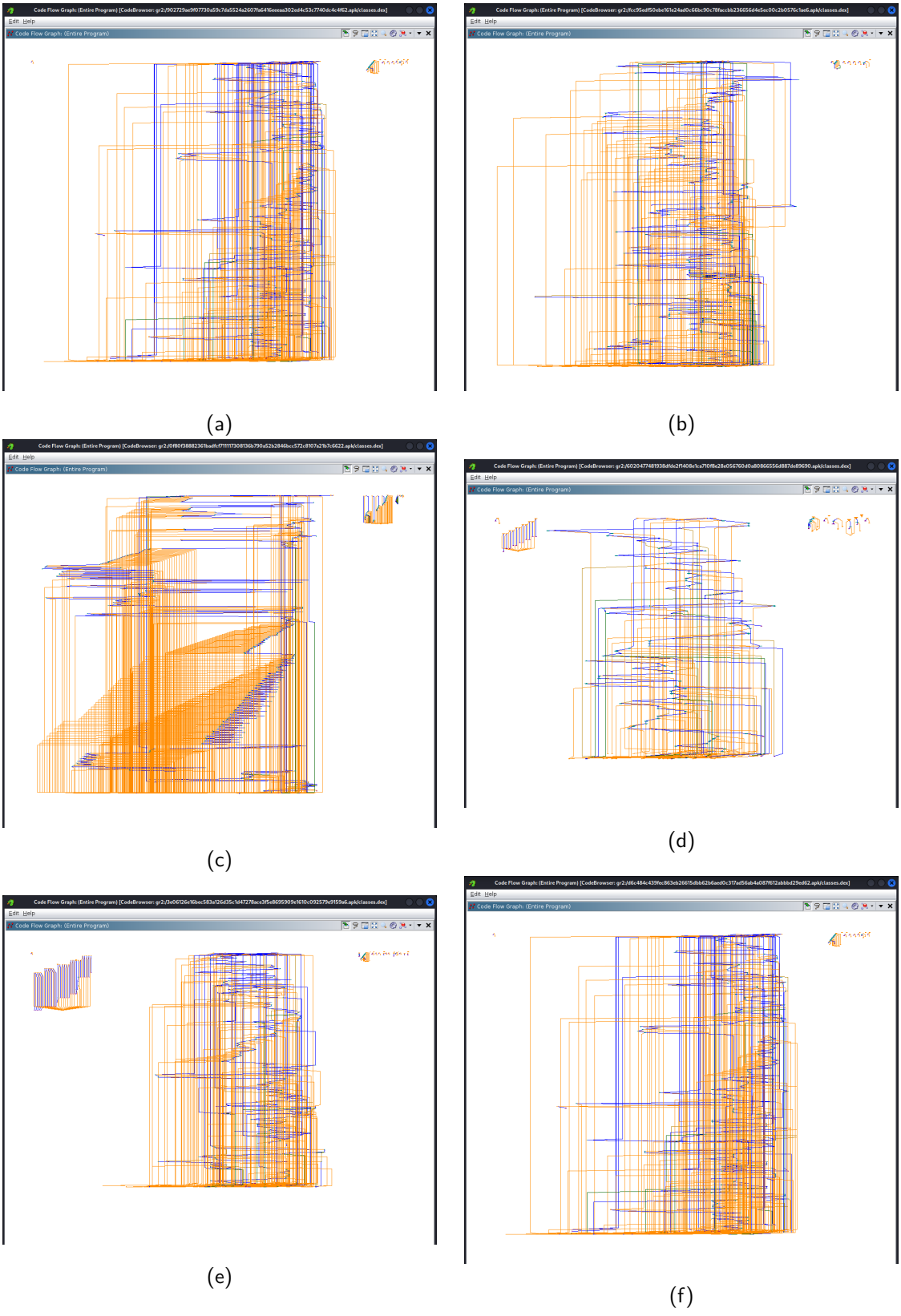
The process of generating graphs involves different tools and techniques, each tailored to extract specific types of information from the APK files.

The following steps outline the graph generation process in Ghidra:

1. Load the APK file into Ghidra.
2. Decompile the binary to access Java-like code.
3. Navigate to the function of interest, or select all code.
4. Choose "Graph" from the action tab.
5. Generate the graph by selecting the desired option (e.g., call graph, block flow graph, or code flow graph).
6. Export the generated graph for further analysis.

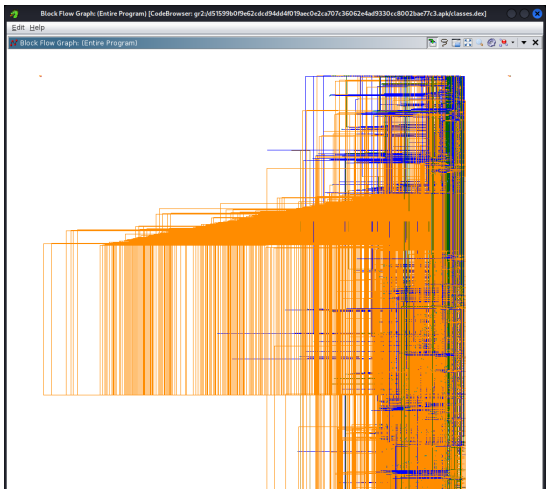
4.3 Examples

The following figures illustrate generated results of call graphs, block flow graphs, and code flow graphs, respectively. These examples showcase how different representations can highlight various aspects of the malware's structure and behavior.

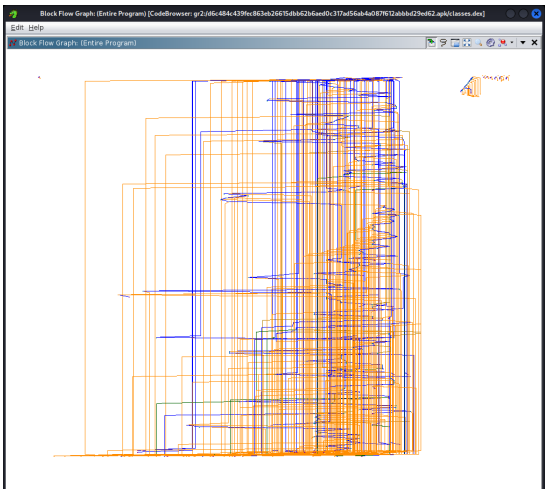


(a) (b) (c) (d) (e) (f)

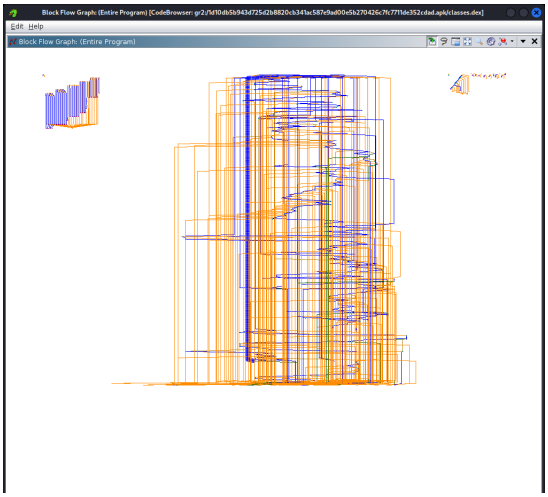
Figure 4.1: Generated code flow graphs.



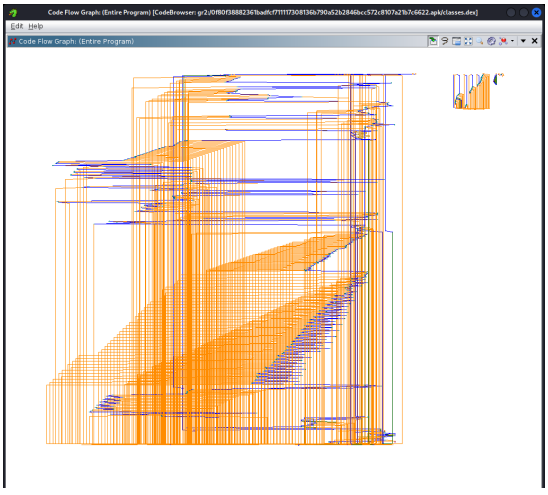
(a)



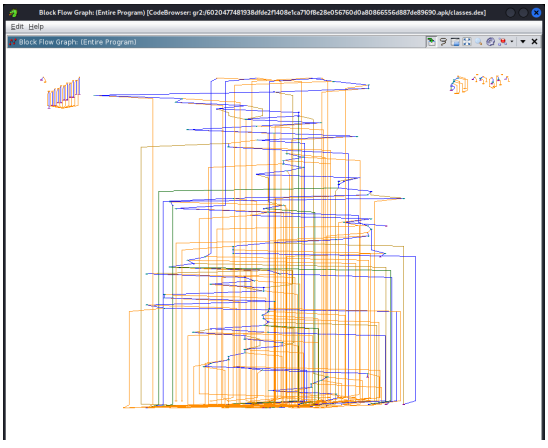
(b)



(c)

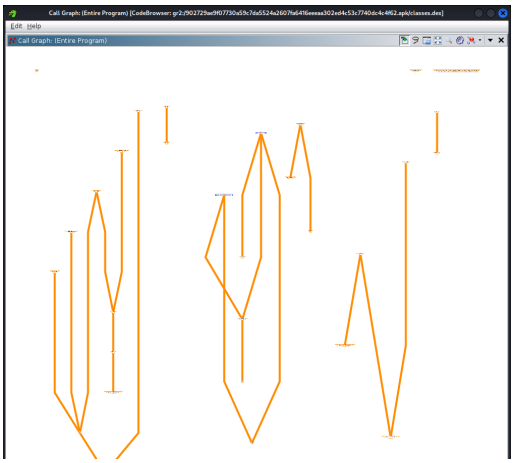


(d)

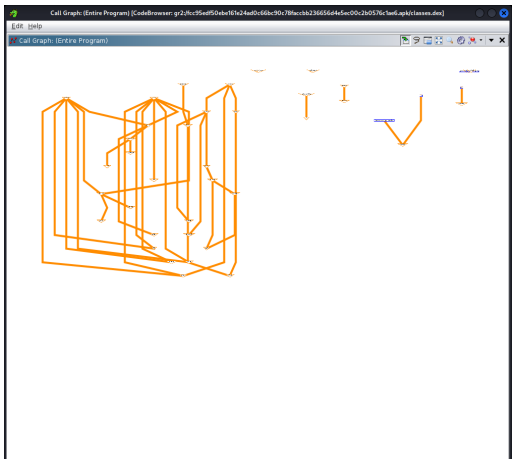


(e)

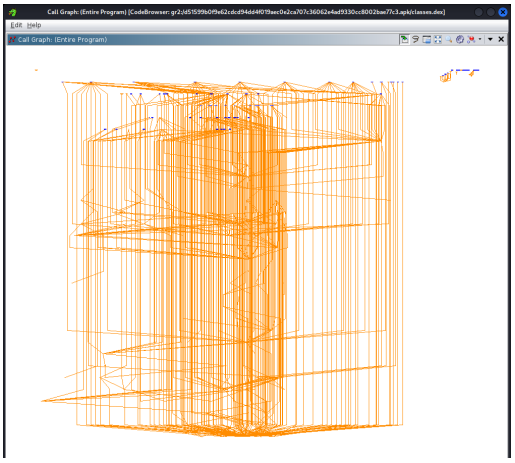
Figure 4.2: Generated block flow graphs.



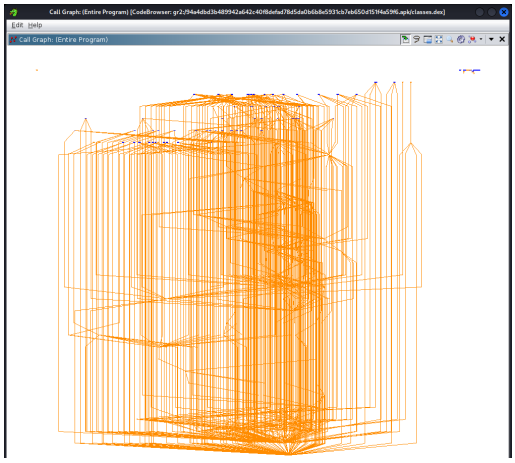
(a)



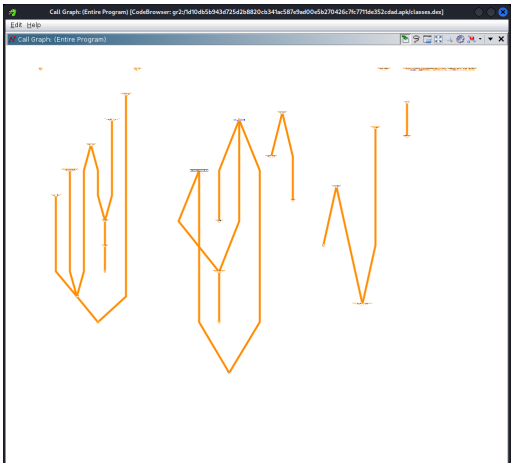
(b)



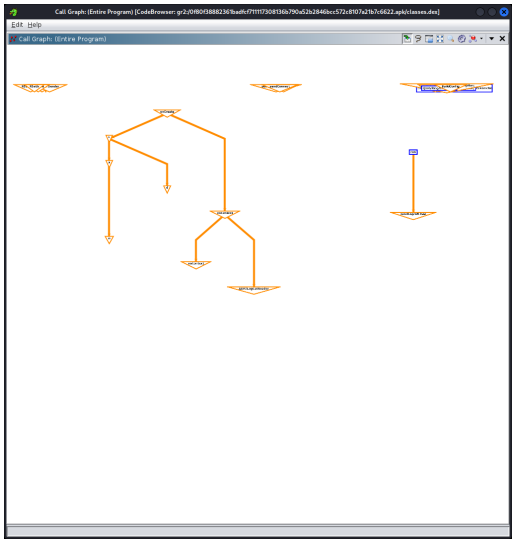
(c)



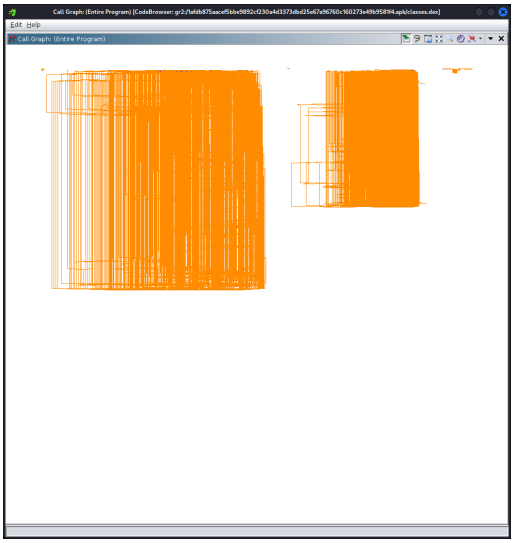
(d)



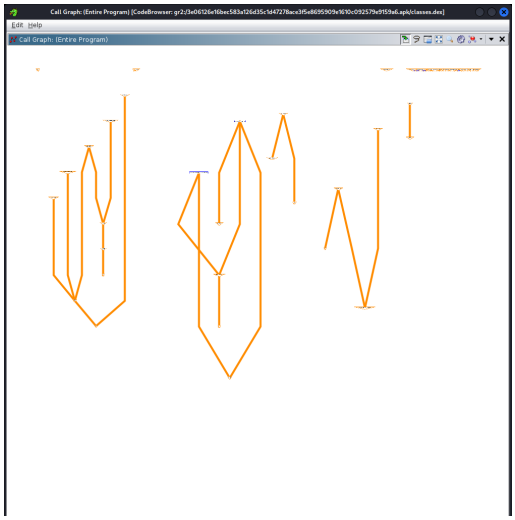
(e)



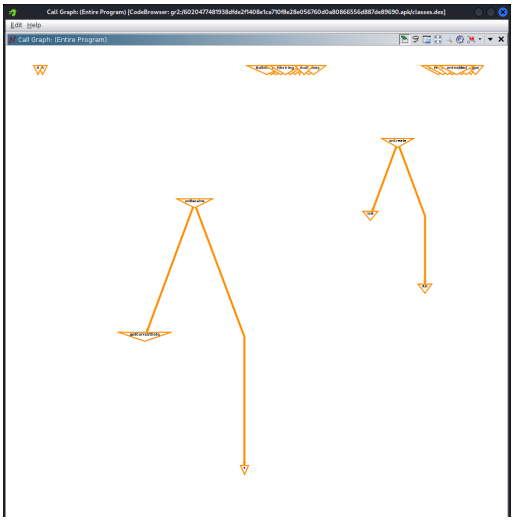
(f)



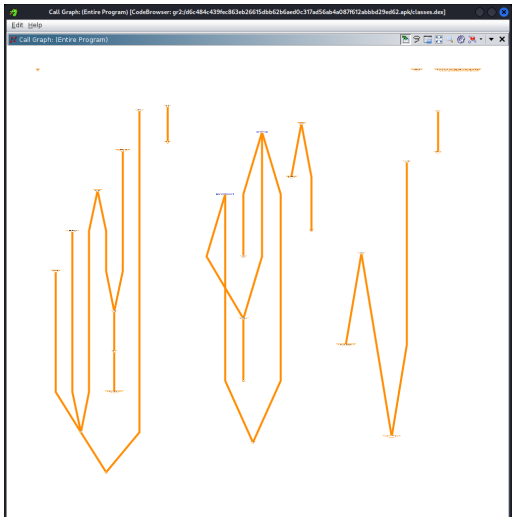
(g)



(h)



(i)



(j)

Figure 4.3: Generated calls graphs.

4.4 Initial Observations

Initial observations from the graph-based analysis reveal several insights about malware behavior.

APKs from the Banking family were chosen for analysis due to their potential to exhibit interesting patterns. Within this family, some APKs display nearly identical call graphs, suggesting shared code or functionality, while others differ significantly, indicating the presence of additional features or obfuscation techniques.

Additionally, block flow graphs and code flow graphs for a single APK are nearly identical, as both emphasize the internal execution pathways of methods. These findings highlight both the similarities and variations in malware behavior within a single family.

4.5 Summary

- Graph-based representations such as call graphs, block flow graphs, and code flow graphs help visualize and understand malware behavior.
- Ghidra is used to generate graphs directly from APK files, including call graphs, block flow graphs, and code flow graphs.
- Examples of the graphs illustrate the structural patterns of the analyzed malware.
- Initial observations show similarities and differences in call graphs among APKs in the same malware family, while block and code flow graphs often align for a single APK.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This research aimed to explore the potential of graph-based analysis for identifying malicious code segments within Android malware. By leveraging static analysis techniques, the project successfully generated multiple types of graphs, including call graphs, block flow graphs, and code flow graphs, to represent the structure and behavior of Android applications.

However, significant challenges were encountered due to obfuscation and anti-reversing techniques employed by malware developers. These methods often resulted in dead code, overly complex call structures, and graphs with an excessive number of nodes. Consequently, the process frequently led to system freezes, memory overload, program crashes, and aborted graph generation.

Through this project, the author gained a deeper understanding of Android's architecture, its components, and their lifecycle, which are fundamental for effective malware analysis. Additionally, the author acquired hands-on experience with various tools, including APKTool, JADX, Ghidra, FlowDroid, and the Soot framework. The knowledge gained from resources like the book *Practical Malware Analysis* further enriched the analysis process and enabled a better understanding of malware behavior and detection methods.

5.2 Future work

While this research achieved its primary objectives, there remain opportunities for further development and improvement.

One potential direction is the automation of graph generation to handle multiple APK files in bulk, which would significantly improve efficiency and scalability. Exploring more advanced tools and techniques to overcome the limitations of current tools and mitigate challenges such as obfuscation and anti-reversing is also crucial.

Furthermore, integrating multiple representations, such as combining call graphs, abstract syntax trees (ASTs), control flow graphs (CFGs), and program dependency graphs (PDGs), could enable more robust comparisons and analyses. Developing a new representation that merges these elements in a cohesive and meaningful way could enhance the ability to identify malicious behaviors and code segments effectively.

References

- Android Developers. (2024). *Android Developer Documentation*. Available at https://developer.android.com/.
- Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st edition, No Starch Press, San Francisco, CA.
- Stone, M. (2020). *Android App Reverse Engineering 101*. Available at https://www.ragingrock.com/AndroidAppRE/.