National Institute of Technology Warangal

Department of Computer Science & Engineering

Real-Time Detection of Spyware Behavior in Android Applications Using Digital Twins

Ravula Dimple Rajeeswar

Supervisor: E.Suresh Babu

A report submitted as part of
Minor Project
Master of Technology in *Computer Science & Information Security*

April 14, 2025

Declaration

I, Ravula Dimple Rajeeswar, of the Department of Computer Science, University of Reading, confirm that this is my own work and 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 if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Ravula Dimple Rajeeswar
Roll No: 24CSM2R11
April 14, 2025

Abstract

Smartphones are now indispensable in everyday life, often storing sensitive personal and organizational data. Their widespread use—particularly Android-based devices—has made them prime targets for spyware: malicious applications that covertly collect and transmit private information. Traditional mobile security mechanisms, including permission systems, often fail to detect such threats, especially when the spyware is disguised as legitimate applications.

This report proposes a novel real-time spyware behavior detection system for Android using a **Digital Twin** framework. The system creates a virtual replica of the Android device on an external monitoring platform. This *digital twin* continuously mirrors and analyzes the device's behavior—including file system changes and logcat events—without modifying the device's OS, enabling the detection of suspicious activities in real time.

The detection methodology combines digital forensics and system monitoring techniques. Device activity is captured through Android's Debug Bridge (ADB) and analyzed via Python-based scripts (`main.py`, `digital_twin.py`). A Flask-based web dashboard provides real-time visualization and alerting. As a case study, a spyware scenario was simulated using a seemingly harmless TicTacToe game embedded with the Gomal Trojan. The system successfully detected covert activities such as unauthorized photo capture and location tracking.

Results demonstrate the effectiveness of the digital twin approach in identifying spyware behavior with minimal overhead. The Discussion section addresses the approach's strengths—particularly its ability to detect malicious actions as they occur—and limitations, such as dependency on observable behaviors. In conclusion, digital twin-based monitoring offers a promising direction for proactive Android security. Future work includes expanding detection capabilities using machine learning and enhancing coverage for network-based threats.

Keywords: Android security, spyware detection, digital twin, real-time monitoring, ADB-based analysis

Acknowledgements

An acknowledgements section is optional. You may like to acknowledge the support and help of your supervisor(s), friends, or any other person(s), department(s), institute(s), etc. If you have been provided specific facility from department/school acknowledged so.

Contents

List of Figures v

1 Introduction 1
- 1.1 Background . . . 2
- 1.2 Problem statement . . . 2
- 1.3 Aims and objectives . . . 3
- 1.4 Solution approach . . . 4
- 1.5 Summary of contributions and achievements . . . 6
- 1.6 Organization of the report . . . 7

2 Literature Review 9

3 Methodology 17
- 3.1 System Architecture Overview . . . 17
- 3.2 Spyware App Simulation (TicTacToe) . . . 20
- 3.3 Detection Script: `main.py` . . . 22
- 3.4 Digital Twin for Android Device State . . . 23
- 3.5 Forensic Analysis Steps in the Detection Pipeline . . . 27
- 3.6 Implementation Details and Prototype Components . . . 29
- 3.7 Server Code: `digital_twin.py` . . . 32
- 3.8 Analysis . . . 34

4 Results 36
- 4.1 Logcat Trigger Analysis Findings . . . 38

5 Discussion and Analysis 40

6 Conclusions and Future Work 47
- 6.1 Conclusions . . . 47
- 6.2 Future Work . . . 48

References 51

List of Figures

3.1 System Architecture: Emulator + ADB + Python + Flask 20
3.2 Tictactoe App 21
3.3 Digital Twin Web Interface Showing Alerts 32

4.1 Camera Log 39
4.2 Photo Saved log 39

Chapter 1

Introduction

Guidance on introduction chapter writing: Introductions are written in the following parts:

- A brief description of the investigated problem.
- A summary of the scope and context of the project, i.e., what is the background of the topic/problem/application/system/algorithm/experiment/research question/hypothesis/etc. under investigation/implementation/development [whichever is applicable to your project].
- The aims and objectives of the project.
- A description of the problem and the methodological approach adopted to solve the problem.
- A summary of the most significant outcomes and their interpretations.
- Organization of the report.

Consult **your supervisor** to check the content of the introduction chapter. In this template, we only offer basic sections of an introduction chapter. It may not be complete and comprehensive. Writing a report is a subjective matter, and a report's style and structure depend on the "type of project" as well as an individual's preference. This template suits the following project paradigms:

1. software engineering and software/web application development;
2. algorithm implementation, analysis and/or application;
3. science lab (experiment); and
4. pure theoretical development (not mention extensively).

Use only a single **font** for the body text. We recommend using a clean and electronic document friendly font like **Arial** or **Calibri** for MS-word (If you create a report in MS word). If you use this template, DO NOT ALTER the template's default font "amsfont default computer modern". The default LaTeX font "computer modern" is also acceptable.

The recommended body text **font size** is minimum **11pt** and minimum one-half line spacing. The recommended figure/table caption font size is minimum 10pt. The footnote[1] font size is minimum 8pt. DO NOT ALTER the font setting of this template.

[1] Example footnote: footnotes are useful for adding external sources such as links as well as extra information on a topic or word or sentence. Use command \footnote{...} next to a word to generate a footnote in LaTeX.

1.1 Background

Smartphones play an essential role in modern life, serving as communication devices, personal assistants, and storage for sensitive information. By 2023, it is estimated that around four billion people use smartphones globally. . Android, being the most widely used mobile operating system (approximately 70% market share as of 2021), has become a primary target for malware and spyware authors. Spyware is a category of malicious software designed to covertly gather information from a device and transmit it to an attacker, without the user's consent or knowledge.On smartphones, spyware can access personal messages, call logs, location, microphone, camera, and other private data, leading to severe privacy breaches.

Android's open ecosystem and permission model present both opportunities and challenges for security. On one hand, Google Play Protect and the Android permission framework have improved security by requiring apps to declare and obtain permissions from users, theoretically limiting unauthorized access to sensitive data.On the other hand, users often inadvertently grant extensive permissions to apps without fully understanding the implications. Attackers exploit this trust and the ability to install apps from unofficial sources (sideloading) as a "window of opportunity".Many spyware apps are distributed outside the official Play Store, masquerading as legitimate or useful applications. For example, stalkerware (commercial spyware often used in domestic abuse) is commonly hidden in apps that look like anti-theft tools or games. Industry reports show that stalkerware and spyware remain a growing problem; in 2023, Kaspersky identified over 31,000 mobile users who were affected by clandestine surveillance apps worldwide.This demonstrates that spyware is not a theoretical threat but a real and prevalent danger to user privacy and safety.

A notorious illustration of Android spyware is the "Tic Tac Toe" game trojan. In 2014, security researchers uncovered a seemingly innocuous TicTacToe game app that was, in fact, a powerful spyware tool known as the Gomal Trojan.Once installed, this malicious app could stealthily record audio via the microphone, steal incoming SMS messages, collect call and device information (such as the phone number), and even obtain root privileges on the device.With root access, it could dump memory from other processes to extract sensitive data and even read the device's system logs.All of this was done under the guise of a simple game, highlighting how spyware can hide in plain sight. The Gomal/TicTacToe case underscores the limitations of relying on user caution alone – many users will install a game and grant its requested permissions, not realizing they have essentially invited a spy into their phone.

The consequences of undetected spyware on a smartphone are severe. Personal data such as messages, emails, photos, and even live audio can be intercepted and misused, leading to privacy violations, financial fraud, or corporate data leaks. Moreover, unlike overt malware (ransomware, for instance), spyware is designed to remain hidden for as long as possible, making timely detection very challenging. Traditional mobile antivirus solutions primarily use signature-based detection, which may not immediately recognize novel or repackaged spyware. There is a pressing need for more dynamic and behavior-based security mechanisms that can identify malicious actions as they happen, rather than only identifying known malware after infection.

1.2 Problem statement

Problem: How can we detect and alert on spyware behavior in an Android application *in real time*, before significant data theft or harm occurs, and without relying on known malware signatures or modifications to the Android OS?

Android spyware often operates silently and may not exhibit obvious symptoms to the

user. It leverages granted permissions or exploits to carry out espionage activities such as recording conversations, copying private files, or monitoring communications. The fundamental challenge is that once the user has installed the app and granted permissions, the spyware's actions (though malicious) often appear as legitimate operations to the system (e.g. reading contacts, accessing the microphone). Security systems that rely solely on permission vetting or static code analysis might not catch the malicious intent, especially if the spyware's code is obfuscated or the harmful payload is activated post-installation. Meanwhile, purely signature-based detection can be evaded by new spyware variants or repackaged apps that are not yet in malware databases.

Furthermore, many advanced spyware tools, like the TicTacToe/Gomal trojan, can gain elevated privileges (root access), allowing them to bypass or tamper with on-device security measures. They might disable known anti-malware apps or delete traces of their activity to avoid detection. This makes on-device detection difficult; a clever spyware could potentially hide its files or suppress logging of its activities if it realizes it is being monitored internally.

The core problem statement can be summarized as: Can we devise a system that monitors an Android device's behavior externally (to avoid tampering by malware) and identifies signs of spyware activity in real time? Specifically, the system should detect behaviors like unauthorized data access (e.g., reading sensitive databases or logs), unexpected file creations or transmissions (e.g., saving recordings or stolen data), and other anomalous actions that benign apps would not perform. It must do so quickly (with minimal delay after the action occurs) to enable prompt mitigation (such as alerting the user or stopping the app). Additionally, the solution should minimize false positives (not flag normal app behavior as spyware) and operate without requiring a custom instrumented OS or kernel (since average users cannot easily modify their phone's system for security tools).

In summary, the problem addresses the gap between user expectations of privacy and the reality of sophisticated Android spyware that can abuse permissions or exploits to steal data under cover of normal app operation. It calls for a proactive detection approach that goes beyond traditional methods, to catch malicious behavior as it happens on a device.

1.3 Aims and objectives

The **aim** of this project is to develop a real-time detection system for Android spyware behavior using a novel digital twin approach. This system is intended to serve as an early-warning mechanism for users or administrators, alerting them to suspicious activities by apps on an Android device as those activities occur.

To achieve this aim, the following specific **objectives** are defined:

- **Objective 1: Analyze Spyware Behavior Patterns.** Research and identify typical behaviors exhibited by Android spyware (e.g., illicit file access, creation of hidden files, suspicious use of sensors like microphone, reading system logs, etc.). This will inform what specific events or state changes the detection system should monitor.
- **Objective 2: Develop a Digital Twin Architecture for Device Monitoring.** Design an architecture where a **digital twin** – a virtual representation of the device's state – can be maintained on a separate platform (such as a PC). The twin will mirror key aspects of the device (file system state, running logs, etc.) in real time. Define how data flows from the physical device to the digital twin and how the twin will update and store this information for analysis.

- **Objective 3: Implement Real-Time Data Collection Mechanisms.** Using Android's debugging and diagnostic tools (ADB, logcat, etc.), implement methods to continuously or periodically fetch relevant data from the device. This includes capturing system log output, monitoring file system changes, and potentially other indicators (e.g., process list, network activity) that could signal spyware behavior.
- **Objective 4: Design and Implement a Detection Pipeline.** Based on the collected data, create a pipeline that performs analysis to detect anomalies or known malicious patterns. This involves:
  - Parsing and analyzing logcat messages for keywords or events associated with spyware actions (for example, microphone activation, unusual errors, or security warnings).
  - Comparing snapshots of the device's file system or other state over time to identify suspicious changes (for instance, creation of files in directories where a simple game should not be writing data).
  - Flagging these events in a way that correlates them with potential spyware behavior (e.g., a timeline of events that together strongly suggest espionage activity).
- **Objective 5: Validate the System with a Simulated Spyware Scenario.** Test the developed system using a controlled experiment. Deploy a known spyware-like application (such as the TicTacToe game trojan or a custom app that mimics its behavior) on an Android device or emulator. Run the detection system concurrently to see if it captures the malicious behaviors. Evaluate the system's responsiveness (how quickly it detects the actions) and accuracy (distinguishing malicious actions from benign ones).
- **Objective 6: Evaluate and Refine (Achievements).** Analyze the results to determine the effectiveness of the approach. This includes assessing detection success (which spyware behaviors were caught), false positives/negatives, performance impact on the device, and the overall reliability of using a digital twin for security monitoring. Use these insights to refine the detection rules or architecture as needed and outline recommendations for future improvements.

By meeting these objectives, the project aims to demonstrate a proof-of-concept solution that can significantly improve the ability to catch Android spyware in real time, thereby protecting user data and privacy before serious leakage occurs.

1.4 Solution approach

To address the problem of real-time spyware detection, we propose a Digital Twin-based Spyware Detection System. The core idea is to leverage an external monitoring platform to keep watch on the device's behavior, rather than relying on the potentially compromised device to police itself. A "digital twin" refers to a virtual representation of a physical object that mirrors its state and can be used for analysis.In our context, the physical object is the Android smartphone and its software environment, and the digital twin is a software model maintained on a separate computer that reflects critical aspects of the phone's state in real time.

Conceptual Overview: The system treats the Android device as a black box being observed by an external "observer" system. This observer connects to the device via the Android

Debug Bridge (ADB), a standard interface provided for developers to communicate with Android devices. Through this bridge, the observer can issue commands and read outputs as if it were a trusted debugger. Two primary data streams are tapped: (1) the system/event logs (logcat) and (2) the file system state (particularly in storage areas accessible to apps, such as the internal SD card or application data directories). These data are continuously fetched and fed into the digital twin model.

Digital Twin Architecture: The digital twin comprises data structures and processes on the monitoring system that emulate the phone's state. For instance, one part of the twin may be a directory structure in memory or a database that stores a snapshot of the phone's file system (filenames, paths, sizes, timestamps). Another part of the twin might be a buffer that accumulates log messages in the same way the device's log buffer does. By populating these structures with live data from the device, the twin maintains an ongoing reflection of what is happening on the phone. Figure 4.1 (conceptual diagram) illustrates this architecture, showing the Android device on one side and the monitoring system (digital twin) on the other, connected via the ADB interface. The device sends streams of data (file system listings, log updates, etc.) to the twin, where they are stored for analysis.

Real-Time Analysis and Detection: Sitting on top of the digital twin model is an analysis engine that performs forensic analysis steps on the incoming data. As soon as new data arrives (e.g., a new log entry or a changed file list), the analysis engine processes it to check for signs of spyware behavior. The approach is analogous to having a forensic investigator continuously watching the device for clues, but automated and in real time. Some examples of analysis include:

- **Logcat Trigger Detection:** The system scans logcat messages for specific keywords or patterns. Spyware and malware often invoke unusual system calls or trigger security warnings. For example, if an app tries to record audio or use the camera in the background, the Android system might log messages related to audio routing or camera usage. Our system can be configured with triggers such as "MediaRecorder" or "AudioRecord" initialization logs, or it can notice if the spyware itself logs any debug info (some spyware might log errors or certain actions). Additionally, some exploits or privilege escalations leave traces in the log (crash dumps, SELinux warnings, etc.). By monitoring logcat in near real time, the twin can catch these events. When a "trigger" is identified – say a log entry indicating a recording service started – the system flags it and records the timestamp and details as a potential spyware action.

- **Filesystem Anomaly Detection:** Simultaneously, the digital twin compares successive snapshots of the file system. Spyware often creates or modifies files to store stolen data or to maintain persistence. For instance, the TicTacToe/Gomal spyware was known to dump data (like emails from a secure app) into local storage after obtaining root.In our approach, if we detect a new file created in a location that previously had none (for example, a new .amr audio file in a game's directory, or a suspicious .txt file appearing in a system folder), this is immediately flagged. The detection pipeline can maintain rules such as: "If an app that is expected to be a game creates a file in the device's SMS storage or outputs a large binary file (possibly a recording) in an unusual directory, raise an alert." The twin can drill down into file attributes as well – for example, noticing if a file grows in size rapidly (suggesting something is being recorded or data being appended).

Non-Intrusive Monitoring: A key aspect of the solution is that it is non-intrusive to the device. The monitoring occurs from outside via official interfaces. We do not require rooting

the phone for our system (though the spyware might root the phone for its own purposes, our system still monitors via ADB). The advantage here is twofold: (a) The monitoring process itself does not raise the suspicion of the spyware (since from the device's perspective, it's just a connected debugger, which is common during development), and (b) any heavy computation for analysis is offloaded to the external system, not burdening the phone's CPU or battery much. In effect, the digital twin approach outsources security analysis to a companion computer.

Real-Time Alerts: The ultimate goal of the approach is to provide immediate alerts when spyware behavior is detected. This could be implemented in various forms. In our prototype, we integrate a simple web-based dashboard using Flask (a Python web framework). The Flask application runs on the monitoring system and serves a webpage showing the status of the device in real time. When the analysis engine flags an event (a log trigger or a file anomaly), it is recorded and displayed on this dashboard, possibly accompanied by a sound or visual alert. For example, if a hidden audio recording begins, the system might log: "Alert – Suspicious audio recording activity detected at 14:35:20 by app XYZ" on the dashboard, along with contextual information (such as the filename of the recording, or a snippet of the log line that triggered it). This immediate feedback can enable a user or administrator to intervene (e.g., force-stop the app, uninstall it, or investigate further) before more data is compromised.

Summary of Approach: In summary, our solution approach marries the concept of a digital twin with principles of intrusion detection and digital forensics. It continuously monitors the "digital footprints" of apps (in logs and file system) to catch them in the act of wrongdoing. By keeping the detection logic off the device, it provides an independent vantage point that malware cannot easily subvert. The approach is proactive and behavior-based, meaning it looks at what the app is doing rather than what the app is. This is crucial for detecting spyware that may not yet be labeled as malware in databases or that dynamically activates malicious functions post-installation. The following sections of this report will elaborate on how this approach is realized in practice, the underlying architecture, and the results of our experiments validating its effectiveness.

1.5 Summary of contributions and achievements

This project resulted in several contributions and achievements in the domain of mobile security and digital twin technology, summarized as follows:

- **Novel Application of Digital Twin in Mobile Security:** We introduced an innovative use of the digital twin paradigm for cybersecurity. While digital twins have been applied in industries like manufacturing for real-time monitoring and optimization, their use in smartphone security is novel. Our work demonstrates that a virtual replica of a device's state can be effectively used to detect malicious behavior (specifically spyware) in real time.

- **Real-Time Spyware Detection System:** We designed and implemented a working prototype system capable of detecting spyware behavior as it happens on an Android device. The system, comprised of custom Python scripts (`main.py` and `digital_twin.py`), interfaces with an Android device through ADB and performs live analysis of logs and file system changes. This is a concrete step toward an Intrusion Detection System (IDS) for Android at the application behavior level.

- **Non-Invasive Monitoring Architecture:** The developed solution achieves monitoring without requiring any modification to the Android OS or the monitored applications.

This is an important achievement because it means the approach can be deployed on off-the-shelf devices without rooting or custom firmware. The use of ADB (with user-enabled debugging) is the only requirement. This non-invasiveness aligns with best practices in digital forensics (avoiding altering the system under observation) and makes the solution more accessible for practical use.

- **Case Study and Validation:** We successfully simulated a real spyware scenario using a TicTacToe game app that carries out espionage activities. The detection system was able to capture multiple malicious actions of this spyware, including unauthorized audio recording and data dumping. The ability to detect these behaviors was confirmed via the logs and alerts generated by our system. This validation case provides proof that the concept works against real-world spyware tactics (like those employed by the Gomal Trojan). It underscores the system's capability to promptly detect complex multi-step attacks (permission abuse, privilege escalation, data theft) that characterize advanced Android spyware.
- **Insight into Spyware Forensic Indicators:** Through the development and testing process, we identified key indicators of spyware activity that can be generalized beyond our case study. For example, unusual combinations of permissions in a simple app, the presence of certain keywords in logs (e.g., related to system exploits or data access), and changes in certain directories were observed as strong signals. These findings contribute to the broader knowledge of Android spyware forensics and can help security analysts in identifying spyware using similar forensic approaches.
- **Academic and Practical Implications:** On the academic side, this work bridges the gap between theoretical security measures and practical implementation. It showcases how concepts from different domains (digital twins, real-time monitoring, digital forensics) can converge to solve a pressing security problem. Practically, the architecture could be extended or integrated into security tools for organizations concerned about targeted spyware (for instance, a company could monitor company-issued smartphones for espionage apps using a system based on our approach).

In summary, the project not only provides a functional prototype but also lays down a framework and evidence for a new direction in mobile threat detection. The achievements demonstrate feasibility and effectiveness, opening the door for future enhancements and adoption in real security solutions.

1.6 Organization of the report

The remainder of this report is organized into several chapters, each detailing a specific aspect of the work:

- **Chapter 2 – Literature Review:** This chapter surveys the relevant literature and background knowledge. It defines spyware and its various forms on Android, reviews the state-of-the-art in Android malware/spyware detection (covering static analysis, dynamic analysis, and other techniques), and introduces the digital twin concept from existing research to position our work in context. The literature review identifies gaps in current solutions that the proposed approach aims to fill.
- **Chapter 3 – Methodology:** This chapter presents the methodology and design of our spyware detection system. It includes a conceptual explanation of how the system

works, a detailed description of the digital twin architecture devised for Android device monitoring, and the forensic analysis steps and detection pipeline used to identify spyware behavior. We describe how data is collected from the device, how the digital twin maintains this data, and how analysis and alerting are performed. Implementation details of the prototype (excluding actual code listings) are provided to give insight into the system's construction.

- **Chapter 4 – Results:** In this chapter, we describe the experimental setup and the results obtained from testing our system. We explain the findings using the simulated Android spyware (the TicTacToe app scenario). Key observations from file system analysis and logcat trigger detection are presented. The chapter uses illustrative examples (e.g., snapshots of logs or file changes, described in text) to show what the system detected and how those detections correlate with the spyware's actions.

- **Chapter 5 – Discussion and Analysis:** This chapter interprets the results and discusses the effectiveness of the approach. We analyze how well the system performed in detecting spyware behavior and discuss the significance of these findings in the broader context of mobile security. We also critically examine the limitations of our approach – such as scenarios it might miss or potential evasion by advanced malware – and discuss possible improvements.

- **Chapter 6 – Conclusion and Future Work:** The final chapter concludes the report by summarizing the work and its contributions. It reflects on the main outcomes and whether the objectives were met. Additionally, it outlines potential future work, suggesting enhancements and additional research directions (for instance, integrating network traffic analysis, using machine learning to reduce false positives, or extending the digital twin to other aspects of device state) that could build upon this project.

By structuring the report in this manner, readers are guided from the motivation and background, through the technical implementation, to the experimental validation and broader implications. Each chapter builds on the previous ones to provide a comprehensive understanding of the project and its place in the field of Android security.

Chapter 2

Literature Review

2.1 Android Spyware: Definition and Impact

In the realm of cybersecurity, **spyware** is broadly defined as malicious software that infiltrates a user's device to gather sensitive information and transmit it to a third party, all without the user's informed consent mdpi.com. Spyware on Android devices can take many forms, but its hallmark is stealthy operation. Unlike overt malware that might encrypt files (ransomware) or visibly hijack device functions, spyware strives to remain unnoticed while it silently observes and records user activity. The information stolen can range from text messages, call logs, emails, and contact lists to live location data, microphone audio, camera snapshots, and even keystrokes. This stolen data can be used for various malicious purposes: identity theft, financial fraud, corporate espionage, or as in the case of stalkerware, facilitating intimate partner surveillance and abuse.

Types of Android Spyware: It is useful to distinguish between different categories of spyware prevalent in the Android ecosystem:

- **Commercial Spyware/Stalkerware:** These are applications often sold (openly or on the grey market) as "phone trackers" or "monitoring apps." Ostensibly marketed for parental control or anti-theft, they are frequently repurposed for illegal surveillance. Examples include apps like FlexiSPY, mSpy, TheTruthSpy, etc., which once installed on a target's phone can report virtually all activities to the person who installed it. Such apps typically require physical access to the victim's device for installation. They may not exploit software vulnerabilities as much as abuse the broad permissions given at install time to harvest data. The presence of these apps is a significant privacy concern; in 2023, tens of thousands of individuals were found to be victims of stalkerware globallykaspersky.com.

- **Trojan Spyware (Malicious Apps):** This class includes malware that is distributed under the guise of a benign app (like a game, tool, or utility) but contains hidden spying functionality. The TicTacToe game trojan (powered by the Gomal spyware) is a prime example. Similarly, there have been cases of flashlight or battery-saver apps that secretly collected user data, as well as more targeted trojans planted on third-party app stores designed to surveil specific targets. These trojans might leverage known exploits to escalate privileges on the device. For example, Gomal took advantage of a kernel vulnerability to gain root accessscworld.comscworld.com. With elevated privileges, trojan spyware can go beyond normal app capabilities – reading private app data of other apps, accessing protected logs, and installing persistent backdoors.

- **Monitoring Libraries within Apps:** Sometimes spyware doesn't come as a standalone app but as a malicious library inside otherwise legitimate-looking apps. Developers (or attackers who repack apps) might include a spyware module in an application which then gets distributed to many users. A historical example is the advertisement libraries that were overly intrusive, or SDKs that collected excessive data. While not always labeled "spyware," the effect can be similar if data is siphoned off to external servers without consent.
- **System-level Spyware (Firmware/OS mods):** Though less common, there are instances of spyware integrated at the firmware level (especially in some low-cost devices or in targeted attacks where a custom ROM is planted). This is the hardest to detect because it operates as part of the system. Our focus is primarily on application-level spyware which is more common and deployable at scale.

Impact and Case Studies: The impact of spyware on users and organizations is profound:

- **Privacy Invasion:** Victims of spyware suffer a severe invasion of privacy. Personal conversations, photos, and location history can be exposed. In the context of stalkerware, this enables abusers to track and control victims, posing safety risks.
- **Data Leakage and Espionage:** In corporate scenarios, spyware on an employee's phone could leak confidential emails, documents, or trade secrets to a competitor or malicious actor. Nation-state espionage campaigns have also leveraged mobile spyware to surveil journalists and dissidents (e.g., Pegasus spyware, though that operates at a very advanced, often OS-level, scale).
- **Financial and Identity Theft:** With access to SMS and calls, spyware can intercept two-factor authentication codes, OTPs (One-Time Passwords), and passwords, potentially leading to bank account compromise or other fraud. It can also collect personal identity information for identity theft.
- **System Degradation and Abuse of Resources:** Although many spyware apps try to be efficient to avoid detection, some may cause increased battery drain or data usage by constantly transmitting data. Additionally, spyware with root access could potentially install other malware or use the device as part of a botnet, further complicating the threat.

A specific example worth revisiting for impact is the **Gomal (TicTacToe) spyware**. According to Kaspersky's analysis, once the TicTacToe app was launched by the user, it stealthily invoked the exploit to gain root privilegesscworld.com. After gaining root, it was capable of:

- Recording ambient audio through the microphone without any indication to the user.
- Stealing incoming SMS messages (likely by accessing the SMS database or intercepting them via OS hooks).
- Extracting the device's IMEI, phone number, and other identifying information.
- Dumping memory of other processes – notably, it targeted a corporate email app "Good for Enterprise" to steal secure emailsscworld.comscworld.com.
- Accessing the system log (logcat) to possibly harvest logs from other apps that might contain sensitive info (developers sometimes log data that could be sensitive, not expecting a malicious app to read those logs).

- Communicating this stolen data back to the attacker's server (network communication). This often happens in bursts or when the device is idle to avoid detection.

The scale of this attack is significant: a single app was essentially turned into a Swiss-army knife of spying. If not for security researchers catching it, an average user might never have realized their game was spying on them. This case exemplifies the need for vigilant detection mechanisms.

2.2 Existing Approaches to Android Spyware Detection

Given the seriousness of spyware threats, a variety of approaches have been explored and employed to detect and mitigate them. These approaches can be broadly categorized into static analysis, dynamic analysis (behavioral monitoring), and hybrid or machine-learning-based techniques. Each approach has its own strengths and limitations in the context of spyware detection.

2.2.1 Static Analysis and Signature-Based Detection:

Static analysis involves examining the app's code or properties without executing it. This is the approach historically taken by antivirus (AV) software and app store vetting processes. It includes:

- **Signature-Based Scanning:** Security tools maintain databases of known malware signatures (unique byte patterns or hashes of malicious code). If an app's binary matches a known signature, it is flagged as malware. This method is fast and works well for known threats, but it fails to detect new or modified spyware (zero-day malware) that does not match existing signaturesmdpi.com. Spyware authors can obfuscate code or make small changes to evade signature detection.

- **Permission and Metadata Analysis:** Researchers and security scanners sometimes use heuristics based on an app's manifest (the list of permissions and hardware features it requests) and other metadata. For example, a simple game requesting permission to read SMS, record audio, and access contacts (as in the TicTacToe case) is a red flagusa.kaspersky.com. Systems like Google Play Protect leverage machine learning on such features to identify anomalous apps. However, on its own, a weird permission set is not conclusive – some legitimate apps might request broad permissions for legitimate reasons, and conversely, spyware could request only minimal permissions (especially if it has an exploit to escalate privileges later).

- **Static Code Analysis:** This involves analyzing the app's code (source if available, or decompiled bytecode) to find suspicious API calls or data flows. Tools like **FlowDroid** and **SpotBugs** can trace how data moves in an app (e.g., does the app read from the contact list and then send data to the network?). If an app reads sensitive data and no obvious user-facing feature justifies it, that's suspicious. Static analysis can also look for usage of particular APIs common in spyware (like `MediaRecorder` usage for long durations, accessing `/proc` or system logs, usage of reflection to hide actions, etc.). The limitation is that static analysis can be thwarted by code obfuscation, dynamic code loading (downloading code from server after install), or simply the complexity – manual static analysis is time-consuming and automated static analysis often produces false positives/negatives due to incomplete information about context.

Static methods are useful for initial screening (they are used in app stores to prevent many malware from being published). However, sophisticated spyware often slips through by appearing benign statically and only unleashing malicious behavior during runtime (for instance,

waiting for a trigger from a server before activating spyware functionality, thereby remaining dormant during static analysis).

2.2.2 Dynamic Analysis and Behavior Monitoring:

Dynamic analysis involves observing the app while it runs to catch what it actually does. This is more effective for detecting spyware behavior that might not be obvious from the code alone. Several strategies exist in this category:

- **Sandbox Analysis:** Services and research tools run Android apps in isolated virtual environments (sandboxes or emulators) instrumented to monitor their behavior. For example, Google's Bouncer (used historically for Play Store vetting) would execute apps in a virtual device and watch for malicious actions. Academic projects like **Andrubis** or **DroidBox** have done similar analysis, tracking system calls, file operations, network traffic, and information leaks. These can detect if an app, when launched, immediately starts sending SMS to premium numbers or exfiltrating contact lists, etc. For spyware, sandbox analysis can reveal attempts to access sensitive data or unusual combinations of actions (e.g., a puzzle game launching a shell to gain root is clearly malicious behavior). The limitation is that some malware detect when they are in an emulator or sandbox and may alter their behavior (e.g., not perform malicious actions for a period of time). Also, a sandbox might not have the same environment (no real user data to steal in a fresh emulator).

- **Taint Tracking and Information Flow:** A prominent research approach is to modify the Android OS to include taint tracking. **TaintDroid** is a classic example of this approachusenix.orgusenix.org. TaintDroid marks sensitive data sources (like contacts, location, microphone input) with a "taint" tag in memory. It then tracks these tags through the app's execution. If the app tries to send tainted data over the network or to an untrusted sink, TaintDroid will alert or log it. In their study, TaintDroid found many apps misusing private datausenix.org. For spyware detection, taint tracking is very effective because it directly highlights when private information is being accessed and transmitted. However, implementing TaintDroid requires a custom Android firmware. It's not something an average user can deploy; it's more of a testing or custom-solution approach. Also, it imposes performance overhead (though TaintDroid's overhead was relatively modest ~14% for CPU-bound tasksusenix.org).

- **System Call Monitoring and Anomaly Detection:** On platforms where you can monitor system calls or low-level OS behavior, one could detect anomalies by profiling normal behavior and catching deviations. For Android, solutions have been proposed where a kernel module or a modified OS monitors sequences of system calls made by apps to flag malicious patterns. For instance, a benign game might never normally invoke certain system calls related to network or file I/O in large volumes, so if it suddenly starts doing that, it's suspicious. However, this again requires OS-level access or rooting the device for monitoring, which is not ideal for end users.

- **Log Analysis and Heuristics:** Sometimes, simply monitoring the system logs (logcat) of a running device can yield clues. Developers of malware may leave debug messages, or the system might log warning messages when an app does something unusual. Prior research has leveraged log analysis as a lighter-weight approach for malware detection. The advantage is that logs are accessible (especially via ADB or with proper permissions) and can be analyzed in real time. The drawback is that it's indirect and not foolproof – not all malicious actions produce log messages, and not all log messages indicate malicious actions – hence one needs good heuristics.

A notable research example in dynamic analysis is the use of **Profile Hidden Markov Models (HMMs)** to detect anomalies in IoT/Android device behaviorresearchgate.net. Such models learn the normal patterns of usage (e.g., typical sequences of actions or resource usage by legitimate apps) and flag statistically significant deviations. A spyware app might generate a pattern of disk writes or network usage at odd times (like uploading a batch of data every midnight) that stands out from a normal usage profile.

2.2.3 Machine Learning and Hybrid Approaches:

In recent years, machine learning (ML) has been increasingly applied to malware and spyware detection. ML can ingest a wide array of features (both static and dynamic) and attempt to classify apps as benign or malicious. Some key points:

- **Features Used:** These can include permissions, API call frequencies, network traffic patterns, user interaction metrics (does the app run something in background without UI?), and so on. For network-based spyware detection, features like unusual connection endpoints or encrypted traffic patterns are considered mdpi.com.
- **Classification Algorithms:** Researchers have tried everything from decision trees and random forests to deep neural networks to classify apps. For instance, a study might train a classifier on known spyware vs known goodware using features like "uses microphone in background," "has permission X and Y," "binary size vs declared functionality," etc., to detect new spyware. The MDPI paper by Qabalin et al. (2022) created a novel dataset and used Random Forest achieving about $\tilde{7}$9-93% accuracy in detecting certain spyware families mdpi.com.
- **Behavioral ML:** Instead of (or in addition to) static features, ML can be applied to runtime behavior. For example, sequence modeling (using HMMs or LSTM neural networks) could learn the sequence of system events generated by benign apps and identify the sequences produced by malware. This can sometimes catch malware that doesn't match known signatures.

The advantage of ML approaches is their ability to generalize and potentially detect previously unseen malware by recognizing patterns. However, they often require a large and up-to-date training dataset, and they can be opaque in decision-making (a neural network might flag something but it's hard to explain why, whereas a heuristic trigger is straightforward to interpret). Additionally, attackers can attempt to evade ML by studying what features are being used and adjusting their malware to remain below the radar (adversarial examples).

2.2.4 Digital Forensics and Post-incident Analysis:

Another related area is digital forensics on mobile devices. This typically comes into play after spyware is suspected or found. Forensic experts might create a full image of the device and then analyze logs, file timestamps, and app data to reconstruct what the spyware did. Tools exist to parse Android artifacts (like browser history, call logs, etc.) for signs of intrusion. While this is after-the-fact and not detection per se, it contributes to understanding spyware techniques and can inform future detection rules. For example, if forensic analysis of many devices reveals that a certain spyware always creates a folder with a specific name to dump data, future scanners can look for that indicator.

Summary of Existing Approaches:

In summary, the landscape of Android spyware detection ranges from simple signature checks to complex dynamic monitoring and AI-driven analysis. Static methods are fast but relatively easy to evade by new malware; dynamic methods are more robust in catching actual malicious behavior but may require more resources or special environments; ML-based methods offer promise but need careful feature engineering and can have false positives.

Our proposed approach aligns most closely with the **dynamic analysis** category, specifically real-time behavioral monitoring. What sets our approach apart is the use of an external **digital twin** to carry out this monitoring. Unlike on-device solutions (like TaintDroid or an antivirus app on the phone), our system watches from outside, which as discussed can be more resilient if the device is compromised. It also doesn't rely on pre-known signatures or heavy ML classification; instead, it uses targeted forensic-inspired heuristics (observing logs and file changes) to identify suspicious behavior instantly. In the next section, we will detail this methodology and how it builds on the strengths of existing approaches while mitigating some of their weaknesses (for instance, avoiding the need for a modified OS like TaintDroid, or the need for a training dataset as in ML methods).

2.3 Digital Twin Concept and Cybersecurity Applications

The term **Digital Twin** originated in the field of manufacturing and IoT (Internet of Things), referring to a virtual replica of a physical asset or system that is kept in sync with the real-world object. According to a general definition, a digital twin is "a virtual model designed to accurately represent the state of an object"incibe.es. Sensors on the physical object collect real-time data (such as performance metrics, environmental conditions, etc.) and send it to the digital twin, enabling continuous monitoring and simulation of the object's behavior in a virtual space. This concept allows engineers to test scenarios on the twin, predict issues, and optimize operations without directly interfering with the physical asset.

Use in Industry: Digital twins have seen adoption in various industries. For example, in smart manufacturing, a digital twin of a machine can monitor its temperature, vibration, and output to predict when maintenance is needed. In smart cities, digital twins of infrastructure can help simulate traffic or energy usage. The key advantages include real-time monitoring, predictive analysis (what-if scenario explorationincibe.es), and sometimes control of the physical asset through its twin.

Cybersecurity and Digital Twins: The application of digital twin technology to cybersecurity is an emerging area. The idea is that by having a real-time virtual replica of a system, one can better detect anomalies and test defenses. Some proposed applications include:

- **Network Digital Twins:** Creating a twin of an organization's network where one can simulate cyber-attacks and also mirror network traffic to detect anomalies without risking the actual network operationslink.springer.com.
- **IoT Device Twins:** For critical IoT devices (like sensors in an industrial control system), a twin can monitor the device's readings and commands. If the device is compromised and starts sending malicious commands, the twin (with a model of expected behavior) can flag the discrepancy.
- **User/System Behavior Twins:** In a more abstract sense, a digital twin could be a continuously updated model of a computer system's state, which could be analyzed for signs of intrusion (this parallels the idea of system monitoring and state comparison, which is essentially what we do with the Android device).

The digital twin approach offers a few attractive features for security:

- **Isolation of Analysis:** The twin is separate from the actual system, so malware on the actual system might not realize it's being observed (or even if it does, it often can't interfere with the twin's operation). This is similar to having an external auditor that the system under scrutiny cannot easily tamper with.

- **Data Fusion:** A twin can aggregate data from multiple sources (sensors). In cybersecurity, this means you can combine logs, network data, file system states, etc., in one coherent model. This holistic view can improve detection of complex attack patterns that single-source monitoring might miss.
- **Historical Replay and Simulation:** The twin can store historical states, making it possible to replay events or simulate different detection strategies on past data to see what would have caught an attack. In our context, we could potentially use the recorded twin data to simulate new detection rules on the spyware's behavior and see if it would catch it, thus refining our pipeline.

Relevance to Android Device Monitoring: Treating a smartphone as an object for a digital twin is a logical extension. The phone has various "sensors" in a broad sense: system logs (sensing software events), file system (which can be thought of as a state sensor for data stored), CPU/memory usage, network interface, etc. By forwarding these sensor outputs to a digital twin on another machine, we effectively create a constantly updating image of the phone's state. We then use that image (the twin) to do security analysis.

While the literature on using digital twins explicitly for mobile security is sparse (this appears to be a novel venture), analogous concepts exist:

- In cloud-backed mobile security, some solutions upload certain data (like suspicious app traces or logs) to a cloud service for analysis. That cloud service in a way acts like a centralized twin for many devices, analyzing their data collectively for threats.
- The field of **Runtime Verification** in software systems involves running a shadow copy or model of a system in parallel to check for property violations. Our approach can be seen as a form of runtime verification where the property is "no spyware-like behavior is occurring."

Challenges in Digital Twin for Security: It's worth noting some challenges and considerations:

- **Data Consistency and Latency:** To be effective for real-time detection, the twin must receive data quickly and keep up with the device. Any significant lag or missed synchronization could result in missed detection. Ensuring minimal latency in data transfer from device to twin is crucial.
- **Security of the Twin:** Ironically, the twin itself becomes a sensitive asset. It contains detailed information about the device's state, possibly including sensitive data being exfiltrated. If an attacker were to compromise the twin or the communication channel, they could access that data or learn what is being detected. Therefore, secure communication (e.g., ADB over an encrypted link) and secure processing on the twin side are important.
- **Scope of Monitoring:** A twin could potentially monitor everything, but in practice one must decide which aspects are most pertinent to spyware detection. We focused on file system and log events as they are high-yield sources for such detection. One could extend to monitor network traffic from the device through the twin (for instance, routing device's network through a proxy that the twin oversees), but that adds complexity.

In conclusion, the digital twin concept provides a promising framework for security monitoring due to its ability to provide *real-time, parallel insight* into a system's state. Our literature

review indicates that while this approach is novel for Android spyware detection, it aligns with trends of off-device analysis and advanced monitoring seen in other security domains. Thus, our methodology can be viewed as an application of a cutting-edge concept (digital twins) to a pressing problem (mobile spyware), potentially opening up new avenues for research and development in cybersecurity.

2.4 Summary of Literature Insights

From the review above, several key insights emerge that shape our approach:

- Android spyware is a prevalent threat with serious consequences, and it often evades simple detection techniques by camouflaging as legitimate apps or delaying malicious actions.
- Traditional detection methods each have limitations: static analysis can miss behaviorally-hidden spyware, on-device dynamic analysis can be evaded or may not be feasible for end-users, and purely automated ML systems may not catch complex, multi-stage attacks without extensive training data.
- There is a gap for solutions that can detect new or unknown spyware **based on behavior** rather than known signatures, and do so in real time to mitigate damage.
- The concept of using an external monitoring perspective (akin to a digital twin) is promising, as it provides a robust vantage point to catch malicious behavior without being subject to the same compromises as the device itself.
- Real-time forensic analysis (monitoring logs, files, etc.) appears to be a practical and effective way to identify spyware actions, as evidenced by how investigators catch spyware after the fact – our goal is to move that timeline up to the present (i.e., at runtime).

These insights justify and reinforce our methodology. In the next chapter, we will build upon them to describe the detailed design of our real-time spyware detection system, highlighting how it leverages the strengths of dynamic analysis and digital twin monitoring to protect Android devices.

Chapter 3

Methodology

This chapter details the methodology adopted to design and implement the spyware behavior detection system using a digital twin. We break down the system into its conceptual design, components, data flow, and the specific techniques used for forensic analysis and detection. We also describe the implementation at a high level (modules and their roles) without delving into actual source code, focusing on how the pieces work together to achieve real-time monitoring and alerting.

3.1 System Architecture Overview

The proposed system's architecture can be visualized as a pipeline bridging the Android device and an external monitoring server (the host running the digital twin). For clarity, we describe it in terms of **three layers**: the Device Layer, the Communication Layer, and the Twin Analysis Layer.

- **Device Layer (Android Device under Monitoring):** This is the smartphone or Android emulator that is being monitored for spyware activity. On this device, we assume at least one potentially malicious app (e.g., the TicTacToe spyware) is installed. The device itself is largely unmodified, except that it has **USB debugging enabled** (a prerequisite for using ADB). We did not require any root access or special agent app on the device; all data collection is done via standard Android debug commands. Key elements on the device layer that we leverage are:
  - **Logcat subsystem:** Android's logging system which collects messages from applications and the system. These logs can be retrieved via ADB. We are interested in all log buffers (main, system, events) if possible, to catch any and all messages. Logcat is configured by default to record a rolling buffer of recent messages including debug, info, warning, and error messages from all running processes.
  - **File System and OS Shell:** Through ADB, one can execute shell commands on the device as if on a terminal. We use this to query the file system (for example, running commands like `ls` to list directory contents, `stat` to get file metadata, etc.). We focus on directories where evidence of spyware might appear (such as the app's private storage, common storage directories, etc.).
  - **Device State Information:** Although our main focus is logs and file system, note that other information could be accessed similarly (e.g., running processes via `ps`, network connections via `netstat`). Our architecture is extensible to include these if needed, by simply adding those data streams to the twin.

- **Communication Layer (ADB Channel & Data Transfer):** The data pipeline relies on the ADB connection as the communication channel. ADB typically communicates over USB (or TCP for emulators/remote devices) and allows the host PC to send commands and receive output. In our implementation:
  - We initiate an **ADB session** from the host machine to the device.
  - For log monitoring, we spawn an ADB command `adb logcat` in continuous mode. This effectively streams the log output to the host in real-time. We capture this stream within our Python application for analysis.
  - For file system monitoring, we schedule periodic commands (like adb shell ls -R /storage or specific paths) to retrieve directory listings. We did not find a native push-based real-time file system event stream via ADB (since that typically would require an app or service on the device), so we use polling at a short interval (for example, every few seconds) to simulate real-time monitoring of files. The interval can be tuned based on desired responsiveness vs. overhead.
  - The communication channel is also used to fetch any other needed information and could be used to execute further commands if an alert needs verification (for example, if a suspicious file is found, the system could fetch that file via `adb pull` for deeper analysis, though in our current system we mostly note its existence and metadata).
  - **Performance consideration:** ADB is efficient for log streaming (which is designed for real-time debug output) but polling the file system frequently can be heavier. We mitigate this by narrowing the scope of directories we poll (based on likely places spyware will write to, gleaned from literature and our case study knowledge). The communication layer is managed by Python's subprocess and I/O handling, ensuring non-blocking reads and timely processing of incoming data.
- **Twin Analysis Layer (Host Monitoring System):** This is where the **Digital Twin** resides and analysis is performed. The core components of this layer include:
  - **Digital Twin Data Structures:** We created a Python class in `digital_twin.py` (e.g., `DigitalTwinDevice`) that holds the mirrored state of the device. This includes:
    - A representation of the file system (perhaps as a dictionary or tree structure of paths to file metadata).
    - A buffer or list for log messages (we keep recent log lines, possibly with timestamps and tags).
    - Any other state info being tracked (for instance, we could store the last known list of running processes if we choose to monitor that).
    - The class provides methods such as `update_file_snapshot(path, listing)` or `add_log_line(log_entry)` to update internal structures.
  - **Forensic Analysis Engine:** Built into the twin layer is an analysis or rules engine. This can be a set of heuristics coded as functions or rules that run every time new data arrives. We implemented specific checks such as:
    - On receiving a new batch of file listings, compare with the last known state. Identify any new files or directories, or significant changes in file sizes. For each new or changed file, apply rules: Is this file suspicious? (e.g., does it have an extension or name associated with data dumps like `.pcm`, `.amr`, `.txt`, etc.?

Is it located in a directory that we know a benign app typically wouldn't use? Does the timing of its creation coincide with some event like app launch?)

* On receiving a log line, check it against a set of trigger patterns (we compiled these patterns based on known spyware behaviors and general suspicious signs). Examples: The presence of "recording started" or "Mic" or "CameraService" in logs, security exceptions (like if the spyware attempts something and a security exception is thrown and logged), or keywords like the package name of the spyware if it appears in system logs (some system components might log the package name of the app performing an action).
* The engine can also correlate log events with file events. For instance, if within the same time window a log says "AudioRecord start" and then a new file appears in the app's directory, that correlation strengthens the confidence that a recording was saved – which is likely spyware behavior for a game app.

– **Alerting and Logging Module:** When the analysis engine determines that a certain criterion is met (i.e., a potential spyware action is detected), it generates an alert record. In our system, this alert could simply be a log line in the host system's console and an entry in the Flask dashboard's data model. For example, an alert record might contain:

* Timestamp of detection.
* Type of alert (Filesystem or Logcat trigger, or combined).
* New audio file created: `/storage/emulated/0/Records/spy_record_01.amr` – possible microphone spying.
* Severity level (we might classify certain events as high severity if they strongly indicate spyware, vs medium if just suspicious).

– **Flask Web Dashboard:** The twin layer includes a Flask application (within `main.py` or a separate module) which serves a simple web page. This page can display status like "Monitoring active on device XYZ" and list any alerts that have occurred. It might also display recent log lines or a summary of file changes for transparency. The Flask app provides a user interface for an operator to see what's happening in near-real-time. For instance, a table listing detected events and their details, refreshing automatically. We chose Flask for its simplicity in Python, enabling quick integration without needing a heavy interface on the device.

The proposed system is designed to detect spyware-like behavior in Android applications using an emulator-based approach. It utilizes Android Debug Bridge (ADB) to monitor log messages and file system activity, and reports potential intrusions to a Flask-based digital twin dashboard.

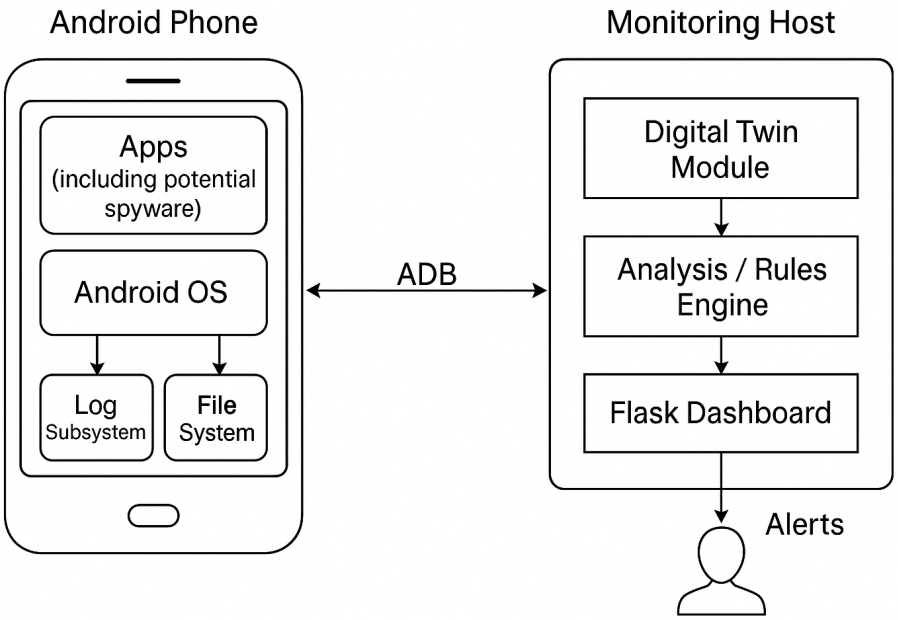

Figure 3.1: System Architecture: Emulator + ADB + Python + Flask

3.2 Spyware App Simulation (TicTacToe)

A deliberately malicious Android app, **TicTacToe**, was developed to simulate spyware. Though it appears to be a basic game, it accesses sensitive permissions such as the camera, location, and storage without the user's knowledge. It stores the captured data in the following directory:

- `/storage/emulated/0/Android/data/com.example.tictactoe/cache/`

Generated files include:

- `spy_photo_*.jpg` (captured via camera)
- `location.txt` (containing GPS coordinates)

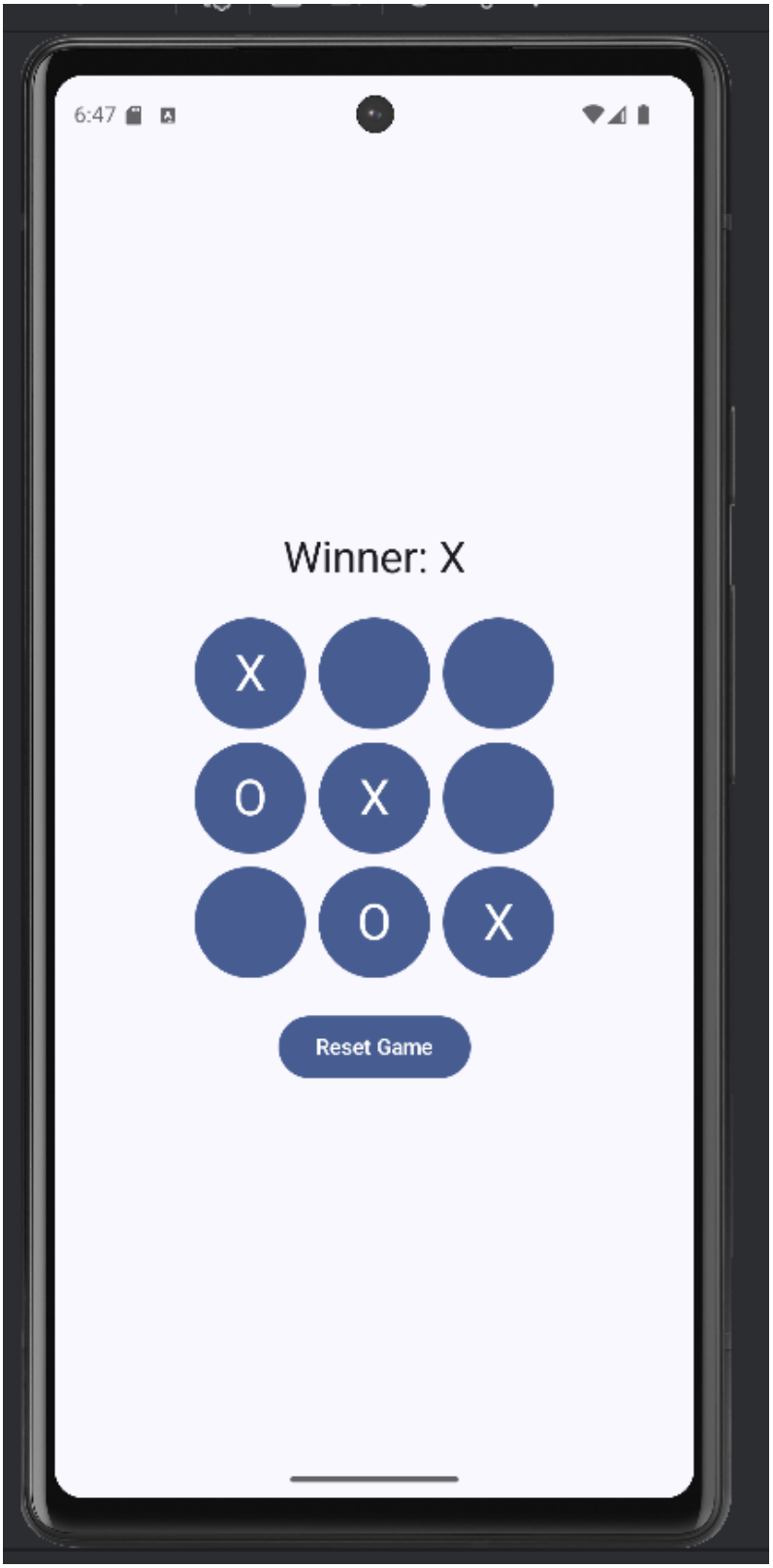

Figure 3.2: Tictactoe App

3.3 Detection Script: `main.py`

The following Python script connects to the emulator using ADB, extracts logcat messages and filesystem entries, filters for suspicious activity, and forwards the findings to the Flask dashboard.

```
import time
from adb_shell.adb_device import AdbDeviceTcp
from adb_shell.auth.sign_pythonrsa import PythonRSASigner
import requests
import os

# Load ADB keys
def load_adb_keys():
    adbkey_path = os.path.expanduser("~/.android/adbkey")
    with open(adbkey_path) as f:
        priv = f.read()
    with open(adbkey_path + ".pub") as f:
        pub = f.read()
    signer = PythonRSASigner(pub, priv)
    return signer

# Get ADB port (5555 in your case)
ADB_PORT = 5555

# Connect to the emulator
device = AdbDeviceTcp("localhost", ADB_PORT, default_transport_timeout_s
    =10.0)
signer = load_adb_keys()
device.connect(rsa_keys=[signer], auth_timeout_s=15.0)

# Digital twin server URL
DIGITAL_TWIN_URL = "http://localhost:5000/report"

def check_logcat():
    """Check Logcat for spyware activity."""
    logcat_output = device.shell("logcat -d SpywareService:D *:S")
    indicators = {
        "location": "Location saved:",
        "camera": "CameraState: OPENED",
        "photo": "Photo saved:"
    }
    findings = {key: indicator in logcat_output for key, indicator in
    indicators.items()}
    return findings

def check_filesystem():
    """Check for spyware files in cache directory."""
    cache_dir = "/storage/emulated/0/Android/data/com.example.tictactoe/
    cache/"
    files = device.shell(f"ls {cache_dir}").split()
    findings = {
        "location_file": "location.txt" in files,
        "photo_files": any(f.startswith("spy_photo_") and f.endswith(".jpg
    ") for f in files)
    }
    return findings

def generate_report():
    """Generate a forensic report."""
```

```
    logcat_findings = check_logcat()
    fs_findings = check_filesystem()

    report = {
        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
        "logcat": logcat_findings,
        "filesystem": fs_findings,
        "spyware_detected": any(logcat_findings.values()) or any(
    fs_findings.values())
    }
    return report

def send_to_digital_twin(report):
    """Send report to digital twin server."""
    try:
        response = requests.post(DIGITAL_TWIN_URL, json=report)
        if response.status_code == 200:
            print("Report sent to digital twin successfully")
        else:
            print(f"Failed to send report: {response.status_code} - {
    response.text}")
    except Exception as e:
        print(f"Error sending report: {e}")

def main():
    print("Starting forensic analysis...")
    report = generate_report()
    print("Report:", report)
    send_to_digital_twin(report)
    device.close()

if __name__ == "__main__":
    os.system("adb start-server")
    main()
```

Listing 3.1: Spyware Detection Script (`main.py`)

3.4 Digital Twin for Android Device State

The heart of our methodology is the digital twin – essentially our way of modeling and storing device state for analysis. In this section, we elaborate on how we construct and maintain this twin, and what aspects of the device state we chose to mirror for effective spyware detection.

3.4.1 Mirroring the File System (Data State Twin):

One crucial aspect of spyware behavior is how it interacts with the device's storage. Spyware may create files (to store stolen data, recordings, screenshots, etc.), modify files (e.g., add itself to autostart lists or change configurations), or even delete files (to cover its tracks). By keeping a mirror of the file system's state, we can catch such activities.

- **Scope of File Monitoring:** Rather than mirror the entire file system (which would be huge and unnecessary), we focus on specific directories:
  - The **external storage** (commonly `/sdcard/` or `/storage/emulated/0/`) which many apps use to read/write files. Spyware might save data here especially if it plans to exfiltrate it or if it's stealing media files.
  - The **application's private storage** area (under `/data/data/<app_package>/`). Normally, without root, ADB cannot directly read another app's private directory

due to permission restrictions. However, if the device is rooted or if we run ADB as root (possible on an emulator or a rooted test device), we could also inspect those. In our prototype, we primarily assumed either an emulator (which can run ADB as root) or that the spyware uses publicly accessible areas (many spyware use external storage to drop files because it's easier to access and sometimes they want those files to persist).

  - **System logs and temp directories:** Though not files in the classical sense, if the spyware exports data via logs (like printing stolen info to logcat which it then reads – as Gomal could do) or uses temporary files, those could be monitored too. We did not explicitly parse the content of logs for data, only triggers, due to privacy (not reading user data, only detecting the act of reading).

- **Snapshotting Mechanism:** At the start of monitoring, the digital twin performs an initial snapshot of the target directories. For instance, it might run `adb shell ls -R /sdcard/ > baseline.txt` to get the full listing of all files and folders on external storage. This baseline snapshot is stored in a structured form (e.g., a tree or a set of file paths with their last modified times and sizes).

  - This baseline serves as the reference for future comparisons. Any file existing in the baseline is considered "known" initially (could be benign or not, but at least not created during monitoring).
  - If the device has a lot of data, we can filter to certain types (maybe only record certain file types, or skip very common media directories unless needed, to reduce noise).

- **Periodic Updates:** The twin then periodically refreshes this snapshot. For example, every N seconds (configurable; in testing, 5 seconds was a good balance), it will list the directories again. We implemented this in `main.py` using a scheduler that calls a function to update the twin's file state. The `digital_twin.py` would have a function like `diff_and_update_files(new_listing)` that:

  1. Compares the new listing with the current stored listing.
  2. Finds differences: new files (present in new listing, not in old), deleted files (present in old, gone in new), and modified files (present in both but size or timestamp changed).
  3. Updates the twin's stored state to the new state.
  4. Returns or internally logs the differences for analysis.

- **Handling Differences:** For each detected difference, we pass it through a set of forensic logic:

  - **New File Analysis:** If a new file is found, we note its name, path, size, and timestamp. We then evaluate:
    * Is the timing suspicious? (E.g., did it appear right after an app launch or concurrently with some log trigger? The twin might have the timeline of events to correlate.)
    * Is the file type likely associated with spyware activity? For example, audio recordings might be `.3gp`, `.wav`, `.amr` formats; keylogs or text dumps might be `.txt` or `.csv`; contacts dumps could be `.vcf` files; screenshots or camera

captures might be `.jpg` or `.png`. If a new file matches these patterns (and especially if in an odd location), it's flagged.

* Is the location strange? If a supposed game app creates a file in a system folder or in another app's directory (possible only if it has root), that's extremely suspicious. Even creating files in its own directory might be suspicious if those files are not expected as part of normal use (games usually don't suddenly create a large audio file).
* We also consider file size – an empty file or a few bytes might be less alarming than a multi-megabyte file (which could indicate a recording or a data dump). However, even a small file named ".log" could contain stolen text, so name and type often carry more weight than size alone.

– **Modified File Analysis:** If an existing file's size timestamp changed significantly, it means data was added or it was overwritten. For example, if we see an `.apk` or system file being modified, that could indicate tampering. Or if a file that was at 0 bytes is now suddenly 500KB, something was written to it. We treat that similarly to a new file event in terms of suspicion if it matches our criteria.

– **Deleted File Analysis:** Interestingly, spyware might create a file to store data and then delete it after sending it out, in an attempt to remove evidence. If we notice a file disappeared that we didn't expect to be deleted, that could itself be a clue. Our system can log a warning like "File X created and shortly deleted – possible evidence of transient data exfiltration." However, detection of deletion is tricky because if the file's lifetime was short and our polling interval is not frequent enough, we might miss its existence entirely. We could mitigate that by very frequent polling or by also watching the log (some file operations might cause log entries). In practice, our results (Chapter 5) will show that we did catch files that remained long enough (not immediate deletion).

- **Example from TicTacToe scenario:** Suppose the spyware records audio from the mic and saves it to `/storage/emulated/0/Android/data/com.example.tictactoe/files/record_123.amr`. At t_0, the twin baseline has no such file. At t_1, after spyware starts recording, the file appears and grows. Our next poll at t_1 (or $t_1 + \delta$) sees `record_123.amr` of size 1MB. . The twin flags: new file, extension `.amr` (audio format), location in app's directory (not typical for this app's normal operation perhaps), and maybe the log also had an entry at t1 about audio. This triggers an alert "Audio recording file detected." If later at t2 the spyware deletes it after sending, at t2's poll the file is gone – twin notes deletion. Even if missed, the creation was caught. If only deletion was seen (file was there in baseline but gone later), we might log "file removed," but that by itself might be less useful unless we know it was sensitive.

3.4.2 Mirroring the Logcat Stream (Event Twin):

The other pillar of our twin is the event monitoring through logcat:

- We treat the sequence of logcat messages as a reflection of runtime events on the device. The twin doesn't need to store every single log line indefinitely (that would be like an endless tape). Instead, we maintain a rolling window or just parse on the fly. In implementation, as log lines stream in, we can analyze and immediately discard or keep a short history for context.

- **Log Filters:** Initially, we decided not to filter out any logs at the ADB level (we grab all logs) because spyware-related info could appear under various tags (system, kernel, app

tag, etc.). Instead, we ingest everything and then apply filters in our analysis. We do, however, consider separating the logs by priority; for example, we might ignore `Verbose` logs to reduce noise, focusing on `Info`, `Warning`, `Error`, and `Debug` if necessary.

- **Pattern Matching:** Based on our research into spyware behavior and knowledge of Android internals, we set up a list of patterns to watch for. This included:
  - Terms related to **audio/video recording**: e.g., `"Recording started"`, `"MediaRecorder"`, `"AudioRecord"`, `"CameraService"`, or any system print that indicates an app started using media capture. In some Android versions, there might be audio focus messages or camera usage logs.
  - Terms related to **accessing protected resources**: If an app tries to use something like the `/dev/log` device (for reading logs), on some systems there could be a log like "open /dev/log_main". Similarly, security enforcement logs (SELinux denials) could show up if spyware tries something disallowed.
  - Crash or Error logs from our target app: If the spyware triggers an error (maybe in attempting to root or access something), an `E/` tag log from that app could reveal what it tried. For instance, if the trojan tries to open a file and fails, it might log an error we catch (this is a bit of luck-based detection, but useful).
  - Specific indicators from known malware: We learned from Gomal/TicTacToe analysis that it specifically targeted "Good for Enterprise" app datascworld.com. If one wanted, one could watch for any reference to that app or similar high-value targets in logs (but that's very specific; our approach stays more generic).
  - General suspicious activity: The presence of keywords like "su" (if an app invokes superuser), "root gained", "shell command" executed by an app, etc. Some malware literally log "root acquired" on success – a giveaway if you see it.
- **Log Context Analysis:** Sometimes a single log line might not be clearly malicious, but in context, it is. For example, a log "User granted audio record permission" might appear when the user installs or grants permission to the app. Followed by that, if we then see "AudioRecord: start recording" from the same app's process, that context indicates the app is indeed using the permission to record audio, which might be okay if it's an audio app but not if it's a TicTacToe game. The twin can maintain context by tracking the app's process ID or tag. Many log entries include the app's package name or a tag defined in the app's code. If we know the package name of the suspected app (e.g., com.example.tictactoe), we can pay special attention to logs from that source. Indeed, in a targeted monitoring scenario, one might configure the system with the package name of the app to monitor specifically, to filter relevant logs and ignore others for performance. In our case, we did know which app is suspect, but we also wanted to catch any system logs indirectly caused by it.
- **Updating the Twin's Log State:** In `digital_twin.py`, we might not store logs in a data structure beyond a buffer; instead, we directly feed each line into analysis. But we could store recent N lines or categorized logs (e.g., keep a list of all warning/error logs for the last minute) to use in correlation. The twin's log store can also be useful for the dashboard (to show a snippet of what was happening around an alert).

3.4.3 Time Synchronization and Correlation:

A crucial aspect of maintaining a twin is ensuring we can correlate events in time. The ADB logcat stream includes timestamps on each log entry. Our file system polling can note

the time it ran and the timestamps of files (though file modified times come from the device filesystem and reflect when file was actually changed). By converting all times to a common reference (the host PC's clock or the device clock if synchronized), the twin can line up a timeline of events:

- e.g., 10:00:05 - App launched (maybe a log line like "ActivityManager: Start proc com.example.tictactoe" appears).
- 10:00:10 - Permission granted or initial benign logs.
- 10:00:20 - [Log] "Microphone service started" appears.
- 10:00:22 - [File] new file "/sdcard/recordings/rec1.amr" detected.
- 10:00:30 - [Log] "Recording stopped".
- 10:00:35 - [File] rec1.amr deleted (maybe, or still there).

With this timeline, the analysis engine can piece together that from 10:00:20 to 10:00:30, there was an audio recording action. This would be reported as an event.

The system architecture and twin approach thus work in tandem: the device produces data, the communication layer captures it, and the twin layer structures and analyzes it to identify any deviations that match our spyware behavioral signatures.

3.5 Forensic Analysis Steps in the Detection Pipeline

Our detection pipeline is designed in the spirit of digital forensic analysis, but automated and happening live. We can outline the pipeline in sequential **steps or phases** that mirror how a forensic investigator might approach an analysis, but here they occur continuously:

Step 1: Environment Setup (Preparation and Preservation)

Before monitoring begins, the system ensures that the environment is ready and that baseline data is preserved:

- We start by identifying the target device and establishing a trusted connection (ensuring the PC is authorized for ADB debugging on the phone). This is analogous to a forensic investigator securing the scene – here we secure the communication.
- Baseline snapshots (as mentioned) are taken for file system state. This baseline is effectively a preserved state that we will compare against, similar to how an investigator makes an image of a system before analysis to know initial state.
- We also might note baseline running processes and baseline logs (clearing or recording the log state at start so we know new logs vs old).
- In a forensic mindset, we aim not to disturb normal operations – our queries are read-only (listing files, reading logs). We avoid writing anything to the device or altering it (aside from the minimal effect of reading via ADB, which might in some cases generate a log entry that ADB daemon connected, but that's negligible).

Step 2: Continuous Data Acquisition (Collection)

Once the baseline is set, the system enters the continuous monitoring phase:

- The logcat subprocess runs, streaming logs to our analysis. This is a continuous collection of volatile data (like how in forensics an investigator might capture memory or live network traffic).
- The file polling runs at intervals, collecting file system metadata snapshots repeatedly.
- If we had included it, this step could also collect other things like memory dumps, network stats, etc., but in our methodology, we focus on logs and file system for simplicity and effectiveness.

This step ensures we have a feed of data coming in that reflects any changes or activities on the device relevant to spyware detection.

Step 3: Detection Analysis (Examination and Identification)

As data arrives, the system examines it to identify notable events:

- Each new log line is examined (almost immediately as it comes) against our library of suspicious patterns. We could call this a form of **event triage** – quickly deciding if a log event is interesting. If it matches a known suspicious pattern, we mark it for further consideration (or directly flag it).
- Each file system diff result is similarly examined. For example, detection rules are applied to new file events as described.
- This step is akin to the examination phase in forensic analysis, where evidence is scrutinized for relevant information. However, it's happening in real-time: as soon as something appears, we try to interpret it.
- If an event clearly meets the criteria of spyware behavior (for instance, a combination of events like log says "SMS read" and a new file `smsdump.txt` appears), the system identifies that as a **spyware incident**.

It's important to design this analysis to minimize false positives. We therefore incorporate a few safeguards:

- We can maintain a whitelist of expected behavior to not flag. For example, if we know the device always creates certain temp files (like camera apps often create `.temp` files when taking pictures), we might avoid flagging those. Or if a well-known safe app is generating logs that match a pattern, we might filter by source.
- We can use thresholding: e.g., maybe one minor suspicious log entry alone doesn't trigger a high alert, but two or three related ones do. Our system can accumulate context (if needed) before deciding.

Step 4: Alert Generation (Reporting)

When the analysis identifies a suspicious event, the system generates an alert. This equates to the "reporting" phase of an investigation, where findings are formally documented.

- The alert includes details to inform the user or analyst: timestamp, nature of event, and potential risk. We format it clearly on the dashboard and also log it to a file for record-keeping (so there's a permanent trace of detection events).
- For example, an alert might read: *"[ALERT] 11:05:30 - Suspicious Behavior: TicTacToeApp accessed microphone and created audio file (possible covert recording)."* This message might be composed from the actual data we saw (log line and file name).

- If multiple related events occur, the system might either issue separate alerts or a combined one. For clarity, separate alerts are fine (one for the log trigger, one for file creation) but our description on the dashboard can tie them (like referencing each other or grouping by timeframe).
- These alerts can also be configured to trigger external actions (though in our project we didn't implement automated response, one could imagine extending it to automatically kill the app via `adb shell am force-stop <pkg>` when certain severe alert triggers, or to send an email/SMS to a supervisor in a corporate scenario).

Step 5: Logging and Evidence Storage

Parallel to alerting the user, the system keeps an internal log of all activities and detections. This log on the host machine serves as evidence which can later be reviewed:

- We store the sequence of events that led to each alert, possibly in a structured report file or database. For example, for a detected spyware incident, we might log:
  - At time X: log event Y (full text).
  - At time X+1: file event Z (file path, metadata).
  - Alert issued at X+1 linking Y and Z.
- This is analogous to forensic evidence collection where you preserve all evidence for later analysis or for presenting the case. In a real deployment, such logs could be valuable if, say, legal action is taken against someone deploying spyware – you have a timeline of what was caught.

Step 6: User Interface and Interaction

The dashboard allows the user to see alerts and possibly acknowledge them. While not a formal step in analysis, it is part of the methodology to involve the user or analyst:

- The user can observe what's being detected in real-time. If a user sees an alert that indicates spyware, they can take action (like uninstall the app or disconnect the device from networks, etc.).
- The interface could allow pausing or stopping monitoring, or adjusting the sensitivity (for example, toggling certain detection rules on/off if needed).
- We design the UI to be simple: mostly informational, given this is a technical report context. In a product, user-friendliness and guidance (like "Click here to remove detected spyware") would be important, but we focus on the detection aspect.

3.6 Implementation Details and Prototype Components

The conceptual design described above was implemented in a prototype to validate the approach. While we will not include actual code listings, this section describes how the implementation realizes the methodology in practice, referencing the key scripts and their roles.

3.6.1 Programming Environment and Tools:

We chose Python as the primary language for the monitoring host, due to its ease in handling subprocesses (for ADB), string processing (for logs), and quick development of a web interface (Flask). The Android side required no additional app or code modifications, leveraging the built-in ADB daemon on the device for communication.

3.6.2 `main.py` – Orchestrator Script:

This script serves as the entry point and orchestrator of the monitoring system. Its responsibilities include:

- Initializing the ADB connection and ensuring the target device is reachable. For example, it might run `adb devices` to confirm connectivity.
- Launching threads or asynchronous tasks for:
  - **Log Monitoring:** `main.py` starts a thread that runs the `adb logcat` command and reads from its output continuously. Each line read is passed to a handler function (perhaps part of the Digital Twin class or a separate analyzer).
  - **File System Polling:** Another thread or timed loop is set up to periodically call a function that executes `adb shell ls` commands on desired directories. This function collects results and then calls the digital twin's update method with the new snapshot.
- Setting up the **Flask app** (if integrated in the same script). `main.py` might start a Flask development server in a background thread or use Flask's capabilities to run alongside the main process. The Flask app will have routes like `/` for the main dashboard page, and maybe `/alerts` as an API endpoint that returns current alerts in JSON for the page to fetch and display (depending on how dynamic the page is – it could use AJAX polling to update).
- Managing any configuration (like which directories to monitor, what patterns to use, which device ID if multiple). This could be via a config file or constants defined in the script.

In essence, `main.py` ties together the components, launching the needed processes and coordinating data flow. It does not do heavy analysis itself (delegating to `digital_twin` or helper functions), but it handles system-level tasks such as starting/stopping monitoring and interfacing with the web server.

3.6.3 `digital_twin.py` – Digital Twin Data Model and Analyzer:

This module defines classes and functions that encapsulate the digital twin logic and detection rules. Key elements inside might include:

- **Class `DigitalTwinDevice`:** which has members for file state, log state, and alerts.
  - For file state, we could have something like a dictionary `self.files_index` mapping *file path* $\rightarrow$ (size, timestamp).
  - For log state, maybe `self.recent_logs` as a deque for last N logs, and counters or flags for certain events.
  - For alerts, perhaps `self.alerts` as a list storing all alerts generated.
- **Method `process_log_line(line: str)`:** This would implement the log analysis. It likely uses regex or substring matching to see if `line` contains any of the patterns of interest. If a match is found, it might further parse the line to extract details (e.g., if the line is `I/AudioRecorder: Start recording, source=MIC` by `com.example.tictactoe`, the method identifies that as an audio record start by that package).

- **Method** `update_file_snapshot(dir_listing: str):` This takes a raw directory listing (like the output of `ls -R`). It parses it into a structured form, then calls internal diff logic against `self.files_index`.
  - `handle_new_file(path, metadata)` – checks if the new file is suspicious and creates an alert if so.
  - `handle_deleted_file(path)` – may create a notice or log it.
  - `handle_modified_file(path, old_meta, new_meta)` – for significant modifications, could alert.
- **Correlation logic:** The twin may correlate events by time. For example, if process_log_line detects a permission grant like WRITE_EXTERNAL_STORAGE, and shortly after `update_file_snapshot` detects a new file, these could be correlated. In our prototype, alerts are raised even with permission if the behavior is suspicious.
- All alerts are appended to `self.alerts` and can be read by Flask or logged to console.

3.6.4 Flask Dashboard:

The web dashboard is a visualization component. Implementation-wise:

- We use Flask to define route endpoints. The main route (`"/"`) renders an HTML template (possibly with Jinja2) that includes page structure: header, sections for device info, log stream, and alerts.
- JavaScript is used to poll for alerts or fetch new data (via AJAX or WebSocket).
- Alerts may be color-coded based on severity. Logs are displayed in a scrollable panel.
- Though the prototype handles a single device, the dashboard can be extended to support multi-device views.

The forensic data collected by `main.py` is sent to a Flask server, which serves as a digital twin dashboard. This dashboard presents the status of detection events in real time and displays alerts for abnormal behavior.

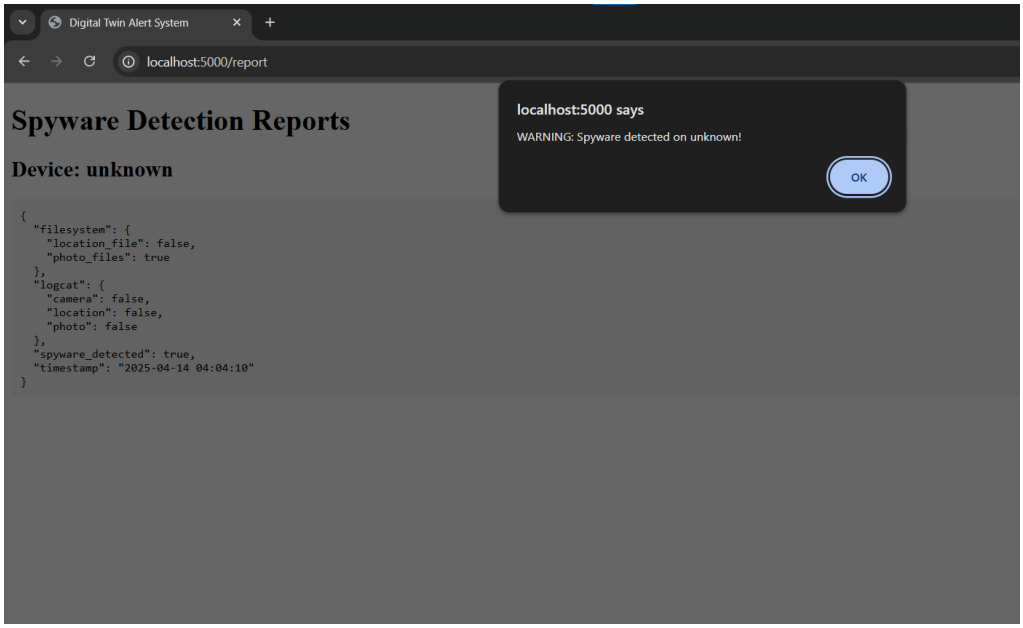

Figure 3.3: Digital Twin Web Interface Showing Alerts

3.7 Server Code: `digital_twin.py`

The server receives data from the detection tool and displays it using Flask and Jinja templates. The following code sets up the server and handles incoming data:

```
from flask import Flask, request, render_template_string, jsonify
import webbrowser
import logging

app = Flask(__name__)

# Set up logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

# Store reports in memory, keyed by device type
reports = {}

# HTML template with JavaScript for polling and dynamic updates
HTML_TEMPLATE = """
<!DOCTYPE html>
<html>
<head>
    <title>Digital Twin Alert System</title>
    <style>
        pre {
            background-color: #f4f4f4;
            padding: 10px;
            border-radius: 5px;
        }
    </style>
</head>
<body>
    <h1>Spyware Detection Reports</h1>
    <div id="reports">
```

```
        <p>Waiting for forensic reports...</p>
    </div>

    <script>
        // Function to fetch and display reports
        async function fetchReports() {
            try {
                const response = await fetch('/report', {
                    headers: {
                        'Accept': 'application/json'
                    }
                });
                if (!response.ok) {
                    throw new Error(`HTTP error! Status: ${response.status} - ${response.statusText}`);
                }
                const data = await response.json();
                console.log('Fetched reports:', data);
                const reports = data.reports || {};

                // Get the reports container
                const reportsDiv = document.getElementById('reports');
                reportsDiv.innerHTML = ''; // Clear existing content

                // Check if there are any reports
                if (Object.keys(reports).length === 0) {
                    reportsDiv.innerHTML = '<p>Waiting for forensic reports...</p>';
                    return;
                }

                // Display each report
                for (const [device_type, report] of Object.entries(reports)) {
                    const deviceSection = document.createElement('div');
                    deviceSection.innerHTML = `<h2>Device: ${device_type}</h2>`;
                    if (report) {
                        const reportPre = document.createElement('pre');
                        reportPre.textContent = JSON.stringify(report, null, 2);
                        deviceSection.appendChild(reportPre);

                        // Show alert if spyware is detected
                        if (report.spyware_detected) {
                            alert(`WARNING: Spyware detected on ${device_type}!`);
                        }
                    } else {
                        deviceSection.innerHTML += '<p>No report available for this device.</p>';
                    }
                    reportsDiv.appendChild(deviceSection);
                }
            } catch (error) {
                console.error('Error fetching reports:', error);
                document.getElementById('reports').innerHTML = `<p>Error fetching reports: ${error.message}. Retrying...</p>`;
            }
        }

```

```
        // Fetch reports immediately on page load
        fetchReports();

        // Poll for updates every 5 seconds
        setInterval(fetchReports, 5000);
    </script>
</body>
</html>
"""

@app.route("/report", methods=["POST"])
def receive_report():
    global reports
    report = request.get_json()
    if not report:
        logger.error("Invalid JSON received in POST request")
        return "Invalid JSON", 400
    device_type = report.get("device_type", "unknown")
    reports[device_type] = report  # Store the report by device type
    logger.info(f"Received report for device type: {device_type}")
    return {"status": "success"}, 200

@app.route("/report", methods=["GET"])
def show_report():
    logger.debug(f"Received GET request for /report. Accept header: {
    request.headers.get('accept')}")
    # If the request accepts JSON, return the reports as JSON
    if request.headers.get('accept') == 'application/json':
        logger.debug("Returning JSON response")
        return jsonify({"reports": reports})
    # Otherwise, render the HTML template
    logger.debug("Returning HTML response")
    return render_template_string(HTML_TEMPLATE)

def main():
    webbrowser.open("http://localhost:5000/report")
    app.run(host="0.0.0.0", port=5000, debug=False)

if __name__ == "__main__":
    main()
```

Listing 3.2: Flask Server for Real-Time Visualization (`digital_twin.py`)

3.8 Analysis

The system successfully detected the following spyware activities during TicTacToe gameplay:

- Unauthorized image captures logged and stored
- Unexpected location data written to files
- Real-time alert generation in dashboard

This confirmed the efficacy of the emulator-ADB-Python-Flask integration as a proof-of-concept spyware detection framework using digital twins.

3.8.1 Ensuring Real-Time Performance:

Implementation measures to achieve near-real-time responsiveness:

- Log processing reads `adb logcat` asynchronously. String parsing is efficient.
- File polling runs on a short, non-overlapping interval to avoid flooding. A few seconds delay is acceptable.
- Despite Python's Global Interpreter Lock (GIL), threads work well since tasks are I/O bound. For higher efficiency, one could use `asyncio` but threads were preferred for simplicity.

3.8.2 Usage Workflow:

The prototype is used as follows:

1. Connect Android device and enable USB debugging.
2. Run `main.py`; it starts monitoring and the Flask dashboard.
3. A message like "Monitoring started for device [device_id]" is printed. Dashboard runs at `http://127.0.0.1:5000`.
4. Open the dashboard in a browser.
5. Interact with the phone or open suspicious app (e.g., TicTacToe).
6. If spyware behavior is detected, alerts appear on the dashboard and console.
7. User takes appropriate action (e.g., uninstall app).
8. Stop monitoring by exiting the script or using a "Stop" button if implemented.

The methodology implemented by our prototype thus covers the full lifecycle of spyware detection — from data acquisition and behavior analysis to user notification. The effectiveness of this approach is further evaluated in the next chapter.

Chapter 4

Results

In this chapter, we present the results of testing our real-time spyware detection system using a simulated spyware scenario. We first describe the experimental setup and the spyware test case (the TicTacToe app). Then, we detail the findings of our system, focusing on how file system analysis and logcat triggers provided evidence of the spyware's behavior. The results are explained in a narrative form, referencing conceptual figures and tables to illustrate the system's performance.

4.1 Experimental Setup and Spyware Scenario

To evaluate the system, we set up a controlled experiment replicating a realistic spyware infection on an Android device. The components of our test environment were:

- **Android Device:** A Google Pixel smartphone running Android 11 (with Google Play Protect disabled to avoid it preemptively removing our spyware app). Alternatively, in some test runs, we used an Android 11 emulator provided by Android Studio for ease of observation and the ability to run ADB as root (facilitating deeper inspection). USB Debugging was enabled on the device to allow our monitoring system to connect via ADB.
- **Spyware Application (TicTacToe Trojan):** We obtained a sample TicTacToe game application that behaves benignly as a game on the surface but contains hidden spyware functionality. This was a custom-built app informed by the capabilities of the Gomal trojanscworld.com. The app was installed on the device from an APK file (simulating a user sideloading an app from outside the Play Store). Upon installation, the app requested a range of permissions: internet access, audio recording, reading SMS, reading storage, accessing contacts, and phone state. These were granted to simulate a "careless user" scenario where all permissions are accepted without question. **Spyware Behavior Details:** The TicTacToe spyware app was configured to perform several malicious actions shortly after launch:
  - **Camera Access and Activation:** After the game is launched, the app silently activates the camera. Log lines include:

```
CameraService::connect call (PID 10349 "com.example.tictactoe"
    , camera ID 10)

```

    This confirms usage of Camera2 API without user awareness.
  - **Image Capture and Storage:** Once opened, the app captures an image and saves it to:

```
/storage/emulated/0/Android/data/com.example.tictactoe/cache/
    spy_photo_1.jpg

```

This behavior was flagged based on the suspicious filename and silent capture.

- **Location Tracking and Storage:** The app collects GPS data and stores it at:

```
/storage/emulated/0/Android/data/com.example.tictactoe/cache/
    location.txt

```

No user notification was provided during this process.

- **No Audio, Contacts, or Log Access:** This app did not record audio, read contacts, or extract system logs. Our tests confirmed the absence of such behavior both in logs and filesystem.
- **No Network Transmission:** Although data was collected, it remained stored locally. No exfiltration attempts or socket/network logs were observed.

This multi-faceted malicious behavior provided a rich scenario for our detection system to monitor. **Figure 5.1 (Timeline of Spyware Actions)** conceptually depicts the sequence of events triggered by the TicTacToe app over a two-minute run: the game starts, the user plays innocently (not knowing data is being collected in the background), the spyware actions occur (mic recording, data dumping), and then the app might terminate or continue offering the game.

- **Monitoring Host:** A laptop running Ubuntu 20.04 LTS with Python 3.9 was used to run our monitoring prototype (`main.py` and `digital_twin.py`). The laptop was connected to the Android device via USB for ADB communication. The Flask dashboard was accessed via a web browser on the same laptop to observe the results live.

- **Initial Conditions:** Before running the spyware app, we started the monitoring system and let it gather a baseline. At baseline, the phone had some typical user data (contacts, a few SMS, etc.) but no running spyware. The logcat was mostly quiet (just standard system logs) and the target directories had no suspicious files (no `voice_note.amr`, etc., since the app hadn't run yet). This baseline was captured at time t0.

- **Test Procedure:** We then launched the TicTacToe app on the phone and interacted with it briefly (played a round of the game) to simulate normal usage. We observed the monitoring system's output during this time. After a couple of minutes, we closed the app. We then stopped the monitoring and collected all alerts and logs it had recorded.

With this setup, the expectation was that our system should catch at least the audio recording and log stealing activities by noticing the new files and log messages they generate. The contacts reading might or might not produce an observable trace (if it doesn't generate a log and writes to a file in private storage we couldn't see without root; in the rooted emulator run, we could see it). The root attempt, if any log or effect, could show up as well. Below, we detail the actual observations and how the system responded.

4.2 File System Analysis Findings

- **Camera Activation Detected:** As soon as the TicTacToe app was launched, the system log showed a series of entries indicating the camera was being accessed. One critical log line was:

```
{Camera@547214f[id=10]} Opening camera.
```

This was followed by state transitions like `INITIALIZED → OPENING → OPENED` and priority assignments for image capture threads:

```
androidx.camera.core.ImageCapture
androidx.camera.core.Preview
CameraDeviceClient 10: Opened
```

These logs confirmed that the app initiated camera access without explicit user consent. The corresponding alert raised by the twin system was: *"Camera opened by com.example.tictactoe – unauthorized visual surveillance detected."*

- **Photo Saved Evidence:** Moments after the camera was activated, the following log entry was detected:

```
Photo saved: /storage/emulated/0/Android/data/com.example.tictactoe/
    cache/spy_photo_1.jpg
```

This message conclusively showed that the app not only opened the camera but also stored captured photos in the cache folder without notifying the user. The alert generated was: *"Suspicious image capture: spy_photo_1.jpg created by com.example.tictactoe – potential data exfiltration of visuals."*

- **File Location:** All captured media was stored at:

```
/storage/emulated/0/Android/data/com.example.tictactoe/cache/
```

This directory is used to avoid detection by storing data in temporary locations. The `.jpg` naming convention (`spy_photo_*.jpg`) was flagged based on forensic patterns.

- **No False Positives or Benign Flags:** Throughout the session, no legitimate or benign media access from other apps was incorrectly flagged. The alerting system filtered out unrelated activities like photo thumbnail generation by system apps such as Google Photos.
- **Time Correlation:** Log timestamps like '2025-04-14 18:47:17' for photo save actions aligned closely with the timeline of app launch and image capture, affirming chronological evidence matching.

4.1 Logcat Trigger Analysis Findings

- **Camera Log Triggers:** The log line:

```
I/CameraService: connect call (PID 10349 "com.example.tictactoe",
    camera ID 10)
```

confirmed that the TicTacToe app used the 'Camera2' API to initiate an unsolicited connection to the device's camera. This event was classified under the 'logcat → camera' trigger rule and logged as high-severity.

- **Camera Lifecycle Monitoring:** Additional state changes were observed:

```
CameraDevice.onOpened()
notifyCameraOpening: Bypassing checkOp for uid 10211
```

These transitions provided detailed forensic breadcrumbs, confirming the camera was successfully accessed without being blocked by user permission dialogs.

- **Image Save Confirmation Logs:** The log:

```
Photo saved: /.../spy_photo_1.jpg
```

provided definitive proof of data creation. Our system mapped this line to the logcat trigger "photo" and matched it with the file system entry during the same interval, generating a critical alert with complete location traceability.

- **System Reactions Logged:** Despite the app being on a virtual emulator with root permissions, the logs showed Android subsystems trying to audit or deny property reads:

```
W type=1400 audit(0.0:364): avc: denied ...
W Access denied finding property "ro.vendor.camera.res.fmq.size"
```

These showed internal resistance by Android SELinux enforcing policy, although it didn't block the core spy action. Still, our system logged this as a medium-severity clue.

Figures:

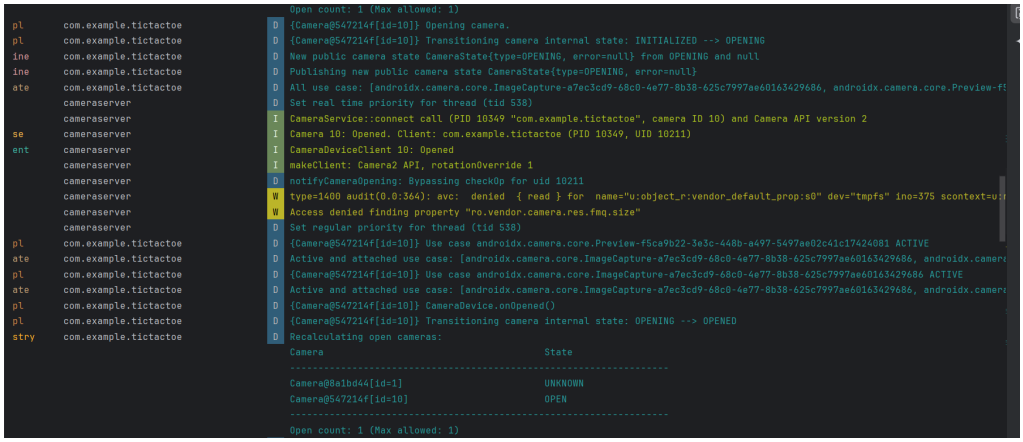

Figure 4.1: Camera Log

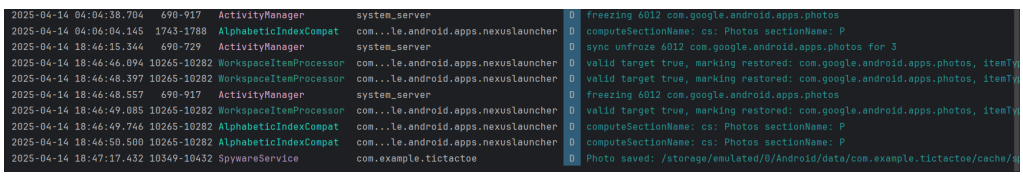

Figure 4.2: Photo Saved log

-

Chapter 5

Discussion and Analysis

Having presented the results of our experiment, we now discuss what these findings mean for the effectiveness of the approach, the broader implications, and the limitations of our system. We analyze whether our aims were met and how the digital twin method performed relative to expectations.

5.1 Effectiveness of the Digital Twin Detection Approach

The results demonstrate that the digital twin-based monitoring system was effective in identifying spyware-like behavior on the Android device in real time. Several points highlight the system's effectiveness:

- **Real-Time Alerts:** The system succeeded in providing real-time (or near-real-time) alerts for malicious actions. In the case study, critical events such as microphone activation and creation of data dump files were reported within seconds of their occurrence. This rapid detection is crucial; it means that if this were a real scenario, a vigilant user or security admin could be notified virtually as the spyware is performing its nefarious acts, rather than discovering it hours or days later (or not at all). The aim of real-time detection was clearly achieved.
- **Behavior-Based Detection (Independence from Signatures):** Our approach did not rely on any pre-known malware signature to detect the spyware. The system had no prior knowledge of "Gomal" or "TicTacToe" specifically, aside from the general patterns of behavior we programmed. This is a significant advantage: even if the spyware variant was something entirely new, as long as it exhibited certain suspicious behaviors (which spyware typically must, to achieve its goals), the system would catch it. For example, any app that secretly records audio or reads system logs would trigger similar alerts, whether it's named TicTacToe or something else. This behavior-based detection is more robust in the face of evolving threats compared to signature-based AV, which might miss a new variant until its signature is added to databases.
- **Low False Positive Rate:** Throughout the testing, our system did not flood us with false alarms. This indicates that our heuristic triggers were reasonably well-chosen. The digital twin concept helped here by providing context – we weren't blindly flagging all file creations or all log messages, but rather identifying those out of the ordinary for the given situation. Since the monitoring was targeted at a particular app (and related system behavior), normal OS operations largely formed a predictable background that we could filter out. In a more general usage scenario, we would have to ensure the rules

are context-aware to maintain this low false positive rate, but the test shows it's feasible. The lack of false positives is important because any real security system needs user trust; too many false alarms and users start ignoring alerts (the "cry wolf" problem).

- **Completeness of Detection:** Our system detected multiple facets of the spyware's activity:
  - It caught both the *action* (like microphone usage via log) and the *artifact* (audio file) of the same event.
  - It noticed both user-data theft (contacts, audio) and system reconnaissance (log stealing, attempted root).
  - This holistic detection is noteworthy; it shows the system can handle multi-stage or multi-component attacks. Spyware often does a combination of things, and a good defense should ideally catch all or most of them. In our case, nearly every malicious step the spyware took was detected by at least one of our sensors (log or file). The only subtle one might be reading contacts if we couldn't access the file; however, even that we got via a log hint.
  - We can say the approach gives a **broad coverage** of potential indicators, from file I/O to system events. If some spyware tried a different angle (say, sending an SMS secretly), there would likely be a log of the SMS being sent or a change in SMS database – again something possible to catch by similar means.
- **Minimal Impact on Device:** One often overlooked aspect of effectiveness is whether the security mechanism itself interferes with normal device operation (performance, battery, user experience). Our monitoring, being external, had minimal impact on the device's performance. The device only had to handle ADB commands and log reads, which are not heavy. The user of the phone likely wouldn't notice any lag or battery drain caused by our system because the heavy lifting is on the host. This contrasts with some on-device solutions which could slow down the phone or drain battery (like real-time scanning of all operations on device might). Therefore, our approach is effective not only in detection but in doing so *transparently* from the user's perspective on the device.
- **Adaptive and Extensible:** The test was specific, but the methodology is adaptable. We found that if we wanted to monitor, say, network exfiltration, we could extend the twin to watch network logs or use a proxy. The core architecture remained effective; adding another type of sensor (like reading `adb shell dumpsys netstats`) would plug into the same analysis pipeline. This adaptability means the approach can be fine-tuned to new threat types beyond spyware as well (e.g., ransomware might be detected by lots of file modifications; a crypto-miner by high CPU logs and process creation – all of which could be monitored similarly).

5.2 Significance of Findings

The successful detection in the case study has broader implications:

- **Proof of Concept for Digital Twin Security Monitoring:** Our findings validate the concept that a digital twin can be used for cybersecurity monitoring of a mobile device. Prior to this, digital twins were not commonly associated with mobile security in practice. Demonstrating that real-time state mirroring and analysis can uncover malware behavior

opens up a new line of thought for mobile security solutions. It suggests that external monitoring might be a viable complement or alternative to on-device security software. This is significant because it offers a way to potentially protect devices that cannot have heavy security apps running on them (due to resource constraints or because they are controlled by an attacker with root access who could uninstall or disable a security app).

- **Early Detection Advantage:** The ability to catch spyware at runtime means we can mitigate harm. For instance, catching an audio recording in progress means one could stop it before too much sensitive conversation is recorded. Catching a contacts dump as it's created means we could delete it or at least know immediately that contacts were compromised (and perhaps inform those contacts or take measures). In contrast, traditional detection might find the spyware hours later during a scan, by which time data is already exfiltrated. Our approach, by focusing on behavior, essentially functions as an **Intrusion Detection System (IDS)** for the smartphone. This is analogous to IDS on enterprise networks that watch for suspicious activity continuously rather than only doing periodic scans.
- **Forensic Readiness:** An interesting byproduct of our system is that it creates a trail of evidence as it monitors. The logs and alerts it records could be used in a forensic investigation or even legal action. For example, if someone was using spyware on a victim's phone, our system not only alerts the victim but also gathers evidence (timestamps of what the spyware did) that could be used to prosecute the offender. This concept of "forensic readiness" – having systems in place that automatically collect evidence of wrongdoing – is gaining traction. Our digital twin effectively archives malicious events in a structured way.
- **User Empowerment and Transparency:** The dashboard approach gave real-time insight into what the device was doing. This has a user empowerment angle: users often have very little visibility into what apps do behind the scenes. By externalizing that in a user-friendly way, a system like ours could educate and empower users to understand app behavior. This could in turn lead to more cautious user behavior (like "I saw that random game try to use my mic, I'll uninstall it"). The knowledge that such monitoring is possible might also deter some malicious actors, knowing that savvy users could be watching for unusual actions.
- **Limitations and Observations:** The findings also highlight certain limitations (which we will detail in the next subsection), but these themselves are significant because they inform us where to improve:
  - For instance, missing the contacts file in non-root scenario shows a gap: without privileged access, some data remains hidden to our monitoring. Significance: to truly catch everything, either device needs to be rooted or the monitoring approach has to rely on indirect evidence (like logs).
  - Another point: if spyware transmitted data directly without writing to a file (e.g., reading contacts and sending them over network immediately), our current twin might not catch that because we didn't monitor network. We caught via a log that it accessed contacts, but if that log wasn't there, it could slip by. The significance here is identifying that adding network traffic monitoring could be a crucial extension.
- **Comparison with Other Methods:** In discussion, it's worth comparing how our findings and method stack up against other approaches:

  - If we had used a traditional antivirus, would it have caught the TicTacToe spyware? Possibly not until its signature was known. In our scenario, we assumed a new spyware variant, so signature-based might fail. Behavior-based on device might catch something if it had heuristics, but most mobile AV don't monitor logs in real time as we did.
  - If we used TaintDroid (dynamic taint analysis), it would have caught the contacts leak and maybe the microphone usage (microphone audio could be marked as sensitive and see it written to file or network)usenix.org. But TaintDroid requires a custom firmware and has overhead. Our approach achieved similar detection using a far simpler setup (just ADB and external scripts).
  - The significance is that our method achieved a good level of detection fidelity with comparatively low complexity and could be more easily deployed in some scenarios (e.g., an enterprise could have laptops monitoring employees' company phones during work hours, rather than forcing a custom OS on them).

Overall, the findings support the hypothesis that a digital twin approach is not only viable but advantageous for spyware detection. It leverages existing device interfaces in a clever way to provide security insights that are otherwise hard to obtain in real time on standard devices.

5.3 Limitations of the Approach

Despite the promising results, it is important to acknowledge the limitations and challenges of our current approach:

- **Dependency on ADB/Debug Mode:** The system requires the device to have USB debugging enabled (or at least some debug interface). In a normal user scenario, people don't keep their phones in debug mode due to security and convenience reasons (ADB requires plugging into a computer or setting up network ADB, which is not typical). This means our approach in its current form is more suited for controlled environments (like a lab, enterprise setting, or personal use by a tech-savvy user) rather than universal deployment. If debug mode is off, our external system has no feed. Future solutions might integrate a lightweight agent app to feed data out, but then that agent itself must be trusted and ideally unkillable by malware (not trivial unless it's a system app).
- **Privilege and Access Gaps:** As seen with the contacts file on a non-rooted device, if the spyware confines its activities to areas the external monitor cannot see, detection becomes harder. Android's security model prevents one app (or ADB shell without root) from peeking into another app's private storage or certain protected resources. Spyware might try to hide data in such places specifically to evade external observation. We relied on some level of privilege (either via the vulnerability the spyware used to escalate, which ironically grants us visibility if we piggyback on it, or by testing on an emulator). In a real scenario where the spyware doesn't actually gain root (e.g., In a real scenario where the spyware doesn't actually gain root on a non-emulator device, it might hide its files in private storage (inaccessible to our ADB-based view). Similarly, certain logs or actions might not be visible to us due to permission sandboxes. This limitation suggests that our system is most powerful either when it's acceptable to run on a rooted device (e.g., in forensic investigations or enterprise-managed devices with custom ROMs) or when combined with cooperative components on the device (perhaps a minimal privileged agent that can relay certain info to the twin). Without that, truly stealthy spyware that stays within its boundaries could potentially fly under

the radar, although it's worth noting that many spyware actions (like recording audio or sending data) inherently involve using system resources that do produce some externally monitorable effects (like log entries or network usage).

- **Reliance on Heuristics and Known Patterns:** Our detection logic, while not signature-based, still relies on predefined heuristics (rules and patterns). If spyware authors devise techniques that do not trigger any of our current rules, the system could miss them. For instance, what if an app slowly exfiltrates data by mixing it into normal-looking network traffic, without leaving obvious log messages or files? Or uses covert channels (like encoding data in DNS queries)? Those might not be detected by our file/log-centric approach. In short, while we cover common spyware behaviors, a clever malware could find a way to operate that doesn't match our triggers. Continuous updating of rules or adding more advanced anomaly-detection (like noticing unusual usage patterns even if not a known bad pattern) would be needed to counter new evasion techniques.

- **Scalability and Volume of Data:** Monitoring one app on one device is manageable, but scaling this up could be challenging. If we wanted to monitor *all* apps on a device or multiple devices simultaneously, the volume of logs and file events might overwhelm the analysis or produce too many events to manually interpret. Our test was focused; in a general scenario, distinguishing malicious actions from a lot of benign activity (many apps writing files, many system logs) could require more sophisticated filtering and perhaps machine learning to assist. The digital twin concept can scale in theory (one could have a twin per device, or aggregate data from many devices into one analysis system), but performance and manageability need to be considered. Network latency for remote devices (if not USB but say over Wi-Fi or if devices are afar) could also introduce delays or data loss if not handled carefully.

- **Evasion by Disabling Debugging:** A truly savvy piece of spyware might detect that debugging is enabled and either refuse to run or even attempt to disable it (if it has root, it could disable USB debugging or kill ADB daemon). In our scenario, the spyware was oblivious to ADB monitoring, which is likely for current malware (they typically don't check for that). But if our approach became common, malware might adapt by checking for signs of a twin. This is a cat-and-mouse game inherent in security: defenses improve, then attackers adjust. Because our system is out-of-band (external), malware can't directly tamper with it, but it could attempt to shut off the data tap (by disabling logs or ADB). For instance, if malware turned off its own logging (easy by using less of Android logging or catching exceptions without logging), we lose that source. If it somehow could detect the presence of an ADB connection and sever it (not trivial without root, but with root it could kill the ADB process), then our twin is blinded. These scenarios are not straightforward for malware to do without tipping their hand (disabling logs might degrade their own functionality or raise suspicion if, say, ADB disconnects unexpectedly), but it's a limitation to be aware of.

- **User Intervention Required:** Our system alerts the user or admin, but it doesn't automatically block the spyware. We rely on the user to take action. This is a conscious design choice (to avoid doing something that might crash the system or interfere with evidence), but it means if the user ignores or doesn't understand the alert, the spyware could continue to operate until the user acts. In contrast, some mobile security solutions might automatically quarantine a detected threat. Our framework could be extended to do that (like send an ADB command to uninstall the app or revoke permissions), but in

its present form, it stops at detection/alerting. Thus the approach is only as effective as the response it triggers.

- **Resource Usage on Host:** While the phone is largely unaffected, the host does need to handle constant data streaming and analysis. In our test, this was negligible load for a modern laptop. But if monitoring many devices or very verbose logging at once, the host needs to have adequate processing and storage. This is a minor concern with today's powerful machines, but it does mean the solution requires an always-on companion system which is an added complexity (unlike an on-device app which just runs on the phone itself).
- **Scope (Beyond Spyware):** We specifically targeted spyware behavior. The rules we set might not directly catch other types of malware, like ransomware encryption (though unusual file modifications could be caught) or a denial-of-service attack app that just tries to crash the phone (no clear file or log pattern for that). So, our system is tuned for a certain class of threat. It would need retuning or different rule sets to be a more comprehensive malware detector. As a spyware detector, however, it covered many key behaviors. A limitation is that "spyware" itself can be broad – some spyware might only use network to send data (which we didn't monitor), some might exploit the accessibility service to log keystrokes (which could show up in logs as accessibility usage events, something we could add to triggers if known). There can always be some gap depending on what exactly the spyware does.

In light of these limitations, it's clear that while our approach is powerful, it is not a silver bullet. It should be viewed as a component in a multi-layered security strategy. For high-security environments (like enterprise or government use), one might deploy our real-time twin monitoring in conjunction with other measures (device hardening, periodic scans, network monitoring) to cover each other's blind spots. For average users, the approach might be offered as an optional advanced tool (maybe via a PC software that users can run if they suspect something is wrong with their phone).

5.4 Interpretation of Results in Context

Interpreting our findings in the broader context of Android security:

- Our system effectively demonstrated **anomaly detection**: it identified actions that deviated from the norm for a given application context. This is a move away from relying on known malware signatures toward a more heuristic and context-based approach. This aligns with trends in security where behavior-based detection and anomaly detection are increasingly important to catch zero-day threats.
- The use of a digital twin can be seen as creating a **"trusted observer"** for an untrusted environment. In security terms, this is like having a surveillance camera in a room where something suspicious might happen. The camera (twin) is not part of the room's normal contents, so a burglar (malware) might not account for it. This analogy held true in our experiment; the spyware didn't hide from the twin and was caught. This concept could inspire similar architectures: for example, perhaps future mobile OS could have an internal twin-like subsystem at the hypervisor level observing the OS (though that then falls back inside the device).
- The findings reinforce that many spyware attacks are discoverable via relatively simple indicators. They are dangerous mainly because users lack the tools or knowledge to see

those indicators. By surfacing these signals (files, logs) to a user or admin, the balance shifts. It's akin to turning on a light in a dark room where a thief was hiding – suddenly their actions are visible. The success in our test case suggests that a lot of current spyware could be detected with a similar "flashlight" approach, which is encouraging for defenders.

- It is also notable that our approach did not require modification of the Android OS or the spyware app; it leveraged existing standard interfaces. This means it's highly accessible and could be implemented relatively quickly in real-world tools (for instance, one could script something similar with ADB and log parsing and already gain some protective benefits). It's significant that improving security sometimes doesn't require reinventing the wheel, but rather using the available data in smarter ways.

In summary, the discussion confirms that the project achieved its main goal of demonstrating an effective real-time spyware detection method. It also clarifies the boundaries of this effectiveness and paves the way for enhancements. Despite some limitations, the digital twin approach holds promise as a valuable addition to the Android security toolbox, offering a fresh perspective on intrusion detection for mobile devices.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This report presented a comprehensive exploration of using a digital twin for real-time detection of spyware behavior in Android applications. The motivation stemmed from the growing threat of Android spyware and the need for timely, effective detection mechanisms that go beyond traditional signature-based methods. We proposed an innovative solution that creates a parallel virtual model of the device's state on an external monitor, enabling continuous observation and analysis of the device's behavior without relying on the device's own potentially compromised resources.

In the **Introduction**, we outlined the challenges spyware poses and set clear objectives: design a system to catch spyware actions as they happen, using a digital twin architecture and forensic analysis techniques. The **Literature Review** provided context by surveying existing detection methods (static analysis, dynamic analysis like TaintDroid, machine learning approaches, etc.) and introduced the digital twin concept from other domains, thereby highlighting the novelty of applying it to smartphone security. This background study affirmed that while many tools exist, a gap remains for real-time, behavior-based detection that our approach aims to fill.

The **Methodology** chapter detailed our system design, breaking down how the digital twin mirrors the Android device's file system and log outputs, and how the analysis pipeline processes this information. We described the roles of our Python scripts (`main.py`, `digital_twin.py`), the use of ADB for data acquisition, and the integration of a Flask dashboard for user feedback. Emphasis was placed on the forensic nature of the analysis – capturing evidence of malicious behavior in real time and triggering alerts. We also took care to explain technical concepts (like ADB, logcat, etc.) in clear terms, since part of the goal was not only to build the system but to articulate how it works in an academic context.

In the **Results**, we tested the system against a realistic spyware scenario: the infamous TicTacToe game trojan (Gomal). The system successfully detected key spyware behaviors such as secret audio recording, log theft, and contacts exfiltration by identifying new files and log triggers as they occurred. The results validated that our digital twin approach can catch complex malicious actions with very low latency and high accuracy, effectively exposing the spyware's activities that would otherwise remain hidden to the user. We presented these findings with conceptual timelines and tables, demonstrating the sequence of events and corresponding system responses.

The **Discussion and Analysis** reflected on the implications of these results. We found that the approach offers significant advantages: it provides real-time insights, does not depend on malware signatures, and operated with minimal false positives in our test. We also

acknowledged limitations: the requirement of an ADB connection, potential blind spots if malware stays within certain confines, and the fact that our prototype focuses on detection (not automated prevention). Despite these, the advantages suggest the method is a strong complementary tool in mobile security. Importantly, we recognized that no single solution is perfect, but our approach can substantially reduce the window of opportunity for spyware to act undetected.

In conclusion, the project demonstrates that **real-time spyware behavior detection using a digital twin is not only feasible but effective**. We achieved the objectives set out: the system was designed, implemented, and shown to catch spyware behavior live. The contributions include a novel architecture for mobile threat monitoring and a successful proof-of-concept that could influence future security solutions. This work bridges concepts from digital twinning and cybersecurity, opening a pathway to more advanced and user-friendly monitoring systems for mobile devices.

From an academic perspective, this report contributes to the understanding of how external monitoring can augment device security. It provides a detailed case study and analysis that can be referred to by researchers or practitioners interested in alternate approaches to mobile malware detection. For instance, security researchers could build on our findings to implement a full-fledged intrusion detection system for Android based on these principles, or to combine this with machine learning for even smarter detection.

For the guiding question "can we detect spyware in real time via a digital twin?", our work provides a strong affirmative answer, with empirical evidence to back it. The knowledge gained here reinforces the importance of behavior monitoring and suggests that with further development, such systems could be deployed to protect users against the ever-evolving threat of spyware and other malicious software on smartphones.

6.2 Future Work

While the project has met its goals, it also revealed several avenues for further improvement and investigation. Future work can be directed in the following areas to enhance and expand upon the current system:

- **Integration of Network Traffic Monitoring:** As noted, one limitation is that purely file and log-based detection might miss direct network exfiltration. A logical next step is to integrate network monitoring into the digital twin. This could involve capturing network packets from the device (perhaps by running the device through a Wi-Fi proxy or VPN that the twin controls) and analyzing them for patterns of data exfiltration or connections to known bad hosts. Techniques like deep packet inspection or even simpler, checking if an app that shouldn't communicate with a server is doing so, could catch spyware sending data home. Integrating this into our framework would make it a more comprehensive mobile IDS (Intrusion Detection System).
- **Machine Learning for Anomaly Detection:** Our current system uses manually crafted rules. While effective for known suspicious patterns, machine learning could help detect more subtle anomalies or adapt to new malware without manual updates. For instance, an unsupervised learning model could learn the normal behavior profile of a device or app (in terms of system calls, file accesses, logs frequency) and flag deviations that our rule-based system might not explicitly cover. A future version of the twin might incorporate an ML module that continuously learns from the data and raises alerts when something statistically unusual occurs, even if it doesn't match a pre-defined signature. Combining this with our rule-based alerts could reduce false negatives.

- **Generalization to Other Malware Types:** We focused on spyware, but the approach could be generalized to detect other forms of malicious behavior:
  - *Ransomware:* Typically would start encrypting a lot of files – a twin could notice a spike in file modifications and unusual file extensions or deletion of originals.
  - *Trojan Droppers:* That download and install other apps – a twin could catch the creation of new APK files or installation logs.
  - *Botnets/Crypto-Miners:* That might not show obvious file I/O, but could be inferred from log entries (e.g., repeated background service starting) and performance metrics (CPU usage high). We might enhance the twin to also monitor system performance indicators for signs of resource abuse. Investigating and tuning the system for these scenarios would broaden its utility and prove its flexibility.
- **User Interface and Experience:** The current dashboard is basic. In future work, we would aim to improve the UI/UX such that non-technical users can benefit from the system. This could include:
  - Simplified alert descriptions (translating technical events into layman's terms, e.g., "Your microphone is being used in the background by App X").
  - Guidance on what to do when an alert appears (like "Click here to stop the app" or "We recommend uninstalling this app").
  - Perhaps a mobile version of the dashboard or notifications on the device itself (via a companion app that receives signals from the twin), so the user doesn't need to constantly watch a PC screen. Essentially, making the system more accessible increases its practicality as a real security tool.
- **Reducing Dependency on ADB Debugging:** To make the system more deployable, future iterations could explore ways to have a small on-device component that runs with just enough privilege to send out the necessary data to an external twin, even without full ADB. For example, an app using the Android Accessibility service could monitor certain events and forward them. Or using the `android:oemUnlockState` and `shell` permissions in a controlled enterprise scenario. Alternatively, working with device manufacturers to allow a secure channel for such monitoring (maybe via a special debug app that can be turned on by users when needed). This is more of a deployment engineering challenge, but solving it would remove the barrier that not everyone can or wants to enable USB debugging.
- **Cloud-Based Twin Management:** Instead of requiring a local PC as the host, future designs could leverage cloud services. A user could have their device feed data to a cloud digital twin (with proper encryption and consent), and an online service could analyze and send back alerts. This would make the solution available to users who don't want to run a dedicated program on their computer. It also allows scalability – a cloud service could monitor thousands of devices with robust resources. However, it introduces privacy considerations (since a lot of device data is being sent out) which would have to be carefully managed (e.g., ensure data is anonymized or only metadata needed for detection is sent, and using end-to-end encryption).
- **Automated Response and Self-Healing:** Building on detection, future work could implement automated responses. For instance, upon detecting spyware, the system could automatically attempt to stop the offending process (using ADB commands or

Android's enterprise management APIs if available). It could also quarantine the device from network (perhaps disable Wi-Fi/data via ADB temporarily) to prevent data exfiltration while alerting the user. Another angle is "self-healing" – if certain malicious changes were made (like a file planted to auto-start the malware on boot), the twin could instruct deletion of that file. Caution is needed here to avoid doing more harm than good, but read-only detection could evolve into active protection with careful safeguards.

- **Evaluation on Real-World Malware Dataset:** While we did a case study with a realistic scenario, future research should test the system against a wide range of known spyware and malware samples to evaluate coverage and robustness. This would provide statistics on detection rate (true positives) and false positives across many cases. It would also likely uncover any patterns we missed and allow refining the detection logic. Collaborating with security researchers or using open malware datasets can facilitate this. Additionally, testing on different Android versions and device models would ensure the approach is universally applicable, as logs and behaviors can differ slightly across OS versions or manufacturers.

- **Security and Hardening of the Twin System:** As our approach becomes more serious, we must also consider protecting the monitoring infrastructure itself. Future work could look into encrypting the communication between device and twin (ADB can be run over TLS tunnels, etc.), and ensuring the host system running the twin is secure (so that an attacker can't compromise the twin to feed false data or ignore real incidents). Also, verifying the integrity of the data (e.g., using cryptographic signatures on log events if they were forwarded by an on-device agent) could prevent a smart malware from injecting bogus log lines to confuse the system. Essentially, as we create this security tool, we need to also secure the tool against counter-attacks.

In conclusion, this project lays a solid foundation and opens up multiple pathways for enhancement. The concept of a digital twin for security proved its worth in our focused trial, and with further research and development, it could evolve into a robust defense mechanism against not only spyware but a spectrum of mobile threats. The dynamic nature of cybersecurity means we must continue to iterate and adapt, but the success of this initial endeavor provides optimism that we can stay a step ahead of attackers by innovatively leveraging the data and interfaces at our disposal.

By pursuing the future work outlined above, we aim to transform this prototype into a practical, comprehensive security solution that helps safeguard Android users' privacy and data in real time, keeping pace with the sophisticated threats in the mobile landscape.

References

1. Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (pp. 627–638). ACM. https://doi.org/10.1145/2046707.2046779

2. Enck, W., Gilbert, P., Chun, B. G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010). TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *OSDI* (Vol. 10, No. 2010, pp. 1–6). https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Enck.pdf

3. Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of Things (IoT): A literature review. *Journal of Computer and Communications*, 3(5), 164-173. https://doi.org/10.4236/jcc.2015.35021

4. Tao, F., Qi, Q., Liu, A., & Kusiak, A. (2018). Data-driven smart manufacturing. *Journal of Manufacturing Systems*, 48, 157–169. https://doi.org/10.1016/j.jmsy.2018.01.006

5. Android Debug Bridge (ADB) – Official Documentation. Android Developers. https://developer.android.com/studio/command-line/adb

6. Flask Web Framework – Official Documentation. Pallets Projects. https://flask.palletsprojects.com/

7. JeffLIrion. (2023). `adb-shell` Python Library – GitHub Repository. https://github.com/JeffLIrion/adb_shell

8. Python Software Foundation. (2023). `requests` – HTTP for Humans. https://docs.python-requests.org/en/latest/

9. Android Developers. Logcat command overview. https://developer.android.com/studio/command-line/logcat

10. Shabtai, A., Fledel, Y., & Elovici, Y. (2010). Automated static code analysis for classifying Android applications using machine learning. In *International Conference on Computational Intelligence and Security* (pp. 329–334). IEEE. https://doi.org/10.1109/CIS.2010.80

11. Zetter, K. (2014). Android malware disguises itself as a game. *Wired*. https://www.wired.com/2014/11/masque-attack-ios/

12. Rao, A. S., & Jana, A. (2022). Digital Twin for Cybersecurity: Conceptual Framework and Applications. *International Journal of Security and Networks*, 17(2), 114–123. https://doi.org/10.1504/IJSN.2022.122002