MALWARE DETECTION – ANDROID (DYNAMIC ANALYSIS)

RESEARCH PROJECT REPORT

Yashsvi Girdhar

BCIT | 2023

LIST OF CONTENT

1. Introduction

Mobile malware poses substantial security risks, impacting both individuals and enterprises by exploiting valuable data stored on mobile devices. Android, holding a 72% market share (Statista, 2023)[1], is a prime target due to its open nature. Cyberattacks, such as mobile banking threats (Kaspersky, 2023)[2], underscore this vulnerability despite efforts by Google and OEMs (Samsung, 2021)[3]. Traditional signature-based detection struggles due to evolving threats, prompting interest in machine learning solutions, especially for zero-day malware (Fedler et al., 2013)[4].

There is vast majority of research in Android malware detection through dynamic analysis, yet a common issue emerges: reliance on inaccessible or outdated tools (predating 2018). Furthermore, a notable trend is the scarcity of open-source initiatives or comprehensive instructions, hindering result reproduction and validation. This study aims to redefine industry norms by advocating for explainable AI over black-box models, aspiring to establish a new standard in Android malware detection.

2. Background

Taxonomy of mobile malware analysis (3 categories: static, dynamic, and hybrid)

	Static Analysis	**Dynamic Analysis**	**Hybrid Analysis**
Implementation	This involves analyzing the code and resources of an app without executing it.	This involves executing the app in a controlled environment to observe its behavior.	This involves combining static and dynamic analysis techniques to achieve better results. Hybrid analysis techniques include using static analysis to identify potential malware and then executing the app in a controlled environment to confirm its behavior.
Techniques	- Opcode analysis - API call analysis - permission analysis It can be used to identify known malware patterns and detect suspicious behavior, such as the use of obfuscation techniques or the presence of hidden functionality.	- System calls - Network traffic - User interactions - Dependency graphs - APIs - IPC (Inter-process communication)	- Code coverage analysis - Taint analysis - Symbolic execution
Benefits	- Low overhead	- Ability to detect zero-day attacks	Combining the strengths of static and dynamic analysis

	- Ability to analyze code-level vulnerabilities - Identify known malware and to flag applications that contain suspicious code or patterns	- Detect malware that is designed to evade static analysis by only activating under certain conditions or by using anti-emulation techniques	
Drawbacks	- Poor documentation - Poor developer habits - Malicious behaviors - Limited visibility into runtime behavior - Limited ability to detect new and unknown malware (because of the use of pre-defined rules)	- Resource and time consuming - Prone to evasion techniques used by sophisticated malware - Hindered by anti-emulation techniques used by malware authors to detect when the app is being executed in a controlled environment - may not capture all possible behaviors of the app - Difficulty in handling polymorphic malware - Difficulty in handling encrypted traffic	- more resource-intensive and time-consuming - Depends on the implementations of techniques, methods used, and the data set. (This point applies to all the analysis methods)

Why Dynamic analysis?

In the realm of cybersecurity, two distinct methodologies are employed to discern potential threats within Android OS. The first involves meticulous scrutiny of the app's content, such as AndroidManifest.xml, .dex files, and so on, known as static analysis, while the second entails the execution of questionable code within a secure environment to observe its behavior—a practice known as dynamic malware analysis.

The pertinence of dynamic malware analysis lies in its capability to unveil novel and undocumented threats, particularly those of the zero-day variety. These elusive threats often evade conventional static analysis techniques, making dynamic analysis crucial in fortifying the security posture of Android phones.

3. Methodology

The research follows a structured path through several phases. It starts with understanding existing work and progresses to selecting and refining features for effective data. Next, predictive models are developed for classification, utilizing insights gained from previous stages. Ultimately, model performance is rigorously evaluated. This strategic approach ensures the creation of accurate and impactful predictive models.

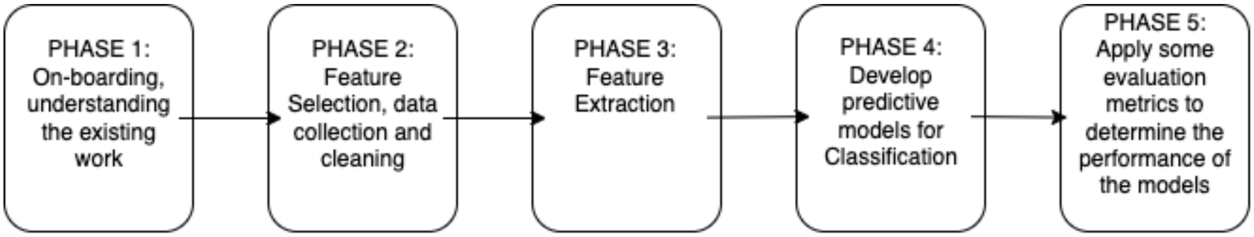

PHASE 1: On-boarding, understanding the existing work
PHASE 2: Feature Selection, data collection and cleaning
PHASE 3: Feature Extraction
PHASE 4: Develop predictive models for Classification
PHASE 5: Apply some evaluation metrics to determine the performance of the models

Fig: Phases of Research

NOTE: We utilized Google Nexus 4 API 31, Android 12.0 rooted emulator to do all the testing/experimenting mentioned throughout in this paper.

Phase 1: Understanding recent approaches towards Dynamic Analysis of android malware detection

Research Questions/Pre-Feature Selection

RQ1: Why system calls?
Sequences of system calls have been recurrently used by anomaly detection systems for security applications in smartphones. All apps interact with the platform where they are executed by requesting services through a number of available system calls. These calls define an interface that allow apps to read/write files, send/receive data through the network, read data from a sensor, make a phone call, etc. Legitimate apps can be characterized by the way they use such an interface, which facilitates the identification of malicious components inserted into a seemingly harmless app and, more generally, other forms of malware .

RQ2: Why Sys-calls over API-calls should be traced?
In the context of Android security, the vast majority of existing frameworks perform API-level tracing (i.e., they aim at obtaining the trace of the APIs invoked by a given app) and use this information to determine whether the app under analysis contains unwanted or malicious functionality. However, previous works have shown that these API-level tracing and instrumentation mechanisms can be easily evaded, regardless of their specific implementation details. An alternative to API-level tracing is syscall-level tracing. This approach works at a lower level, and it extracts the sequence of syscalls invoked by a given app: the advantage is that this approach can be implemented in kernel space and, thus, it cannot be evaded, and it can be very challenging, if not outright impossible, to be detected by code running in user space. However, while this approach offers more security guarantees, it is affected by a significant limitation: most of the semantics of the app’s behavior is lost. These syscalls are in fact low-level and do

not carry as much information as the highly semantics-rich Android APIs. In other words, there is a significant semantic gap.

RQ3: Why network calls should be traced?
In response to the concerning prevalence of mobile malware that targets user information and credentials (Felt et al., 2011)[22], there is a need focus on detecting self-updating malware capable of data theft and unauthorized user surveillance. Thus, a need to monitor and identify significant deviations in the network behavior of mobile applications. This would serve a dual purpose: safeguarding mobile users against data-stealing and user-spying malware, while also enabling the collection and analysis of network traffic patterns. This analysis can be leveraged to enhance cellular infrastructure security by identifying malicious or disruptive applications.

Phase 2: Attempt to reproduce the works of existing tools for dynamic analysis

Reproducing Existing Work

Tools explored	Feature(s)	Year
Twindroid[5]	Syscalls	2022
DroidSpan[6]	SAD profile using syscalls	2020

Results from the tools explored:
1. We were able to successfully run Twindroid and generate syscalls using their shell script.
2. After transitioning the code from Python v2.7 to Python v3, we managed to execute Droidspan's code effectively. Nonetheless, no viable data/logs were produced.

Phase 3: Create our own python scripts[7] to be able to collect syscalls and network calls

During this phase we explored a variety of tools to be able to create our own scripts to collect sycalls and network traffic produced by running an app on a device.

Tools Utilized[12-19]	Purpose
1. adb	ADB, or Android Debug Bridge, is a command-line tool crucial for Android development and debugging. It enables tasks like: • installing and managing apps • accessing device shells • transferring files • capturing screens, and more.
2. aapt	aapt, or Android Asset Packaging Tool, is a command-line utility that is a crucial part of the Android development toolkit. It plays a pivotal role in preparing and managing Android application packages (APKs) and their resources.

[5] https://github.com/RaphaelKhoury/automated-apk-tracing
[6] https://bitbucket.org/haipeng_cai/droidspan/src/master/
[7] https://github.com/maryam-tanha/DynamicAndroidMalwareAnalysis
[8] Diagrams

	• **Resource Handling:** Compiles, packages, and optimizes app resources (images, layouts, strings) into efficient binary format. • **APK Creation:** Packages compiled resources and app code into the final APK for distribution. • **Resource Localization:** Supports different languages and regions by managing localized resource versions. • **Manifest Management:** Merges app manifest with generated data, ensuring accurate app structure. • **APK Signing:** Assists in signing APKs, a requirement for app distribution. • **Dynamic Features:** Creates APKs for dynamic feature modules in Android apps. • **Resource ID Assignment:** Assigns unique IDs to resources for runtime access and efficiency.
3. strace (for syscalls)	Linux command-line utility which enables us to monitor and record system calls and signals generated by a program during its execution.
4. monkey	The Android Monkey tool is a command-line utility for automated testing of Android apps. It simulates random user interactions to stress-test apps and uncover potential issues.
5. tcpdump (for network calls)	Linux command-line tool that captures and analyzes network packets in real-time. It helps diagnose network issues, analyze protocols, and detect security threats by monitoring and displaying network traffic details.
6. openssl	openssl is a versatile command-line tool and library for working with secure communications and cryptography in Unix-like operating systems. We utilized it for: • Generating X.509 certificates and cryptographic keys used in secure communication and authentication. • To test SSL/TLS connections, perform handshakes, and diagnose security-related issues.
7. Emulator (Genymotion and Android SDK AVD)	Genymotion is an advanced Android emulator. It offers superior performance, a variety of virtual devices, customization options, and features like sensor simulation and network testing. It's a valuable tool for efficient app development and testing.
Other Tools Explored	**Purpose**

8. Frida	It is an open-source toolkit that lets you inject code into running apps on various platforms. It's used for dynamic analysis, reverse engineering, and modifying app behavior in real-time. It's JavaScript-based, making it versatile for different platforms and tasks.

Why we chose Monkey Tool?

Most of the existing works uses monkey in some form or the other, i.e., either they directly use monkey to collect data from running the app or use a wrapper built on top of the monkey tool. Android Monkey tool allows us to:

- Automate testing by simulating random user interactions.
- Quickly performs a high volume of interactions for faster testing.
- Test across various device configurations for compatibility.

However, Monkey tool:

- Might miss specific use cases.
- Not ideal for testing precise user flows.

NOTE: There are some other user-guided tools that can be configured better for our use case, instead of using Monkey Tool.[9]

Why we chose to use an Emulator and NOT an actual device?

The decision to employ an emulator rather than an actual device was underpinned by several practical considerations, aligning with the research's objectives and constraints. Initially, our research necessitated a rooted device to facilitate in-depth feature extraction. However, our attempts to root devices, such as Huawei P30 Pro and Samsung A13, encountered roadblocks, as these specific models posed challenges to successful rooting.

[9]

SwiftHand
SwiftHand2
Calabash Android
Appium

Extracting Syscalls:

This Python script, inspired by automated apk tracing, TwinDroid[5] automates the process of running an APK on an Android device, capturing the sycalls being made during the monkey enabled testing of the apk file.

It is achieved by performing the following tasks:

1. Installs an Android application from the input repository using ADB.
2. Retrieves the process ID (PID) of the zygote process.
3. Runs strace on the zygote process, capturing the system calls and their output into a file.
4. Starts an application specified by the package name and runs it for 20 seconds.
5. Simulates user interactions on the application using Monkey for 2000 seconds.
6. Stops the strace process and moves the strace output file to the output repository.

Pseudo-code for extracting syscalls:

```
function extractSyscalls ():
    adb_install(repository_path + '/' + apk_name)

    zygote_pid = adb_run_shell_command("pidof zygote")

    strace_output = start_strace_capture(zygote_pid)
    wait_for_seconds(20)
    stop_strace_capture(strace_output)

    adb_start_app(package_name)
    wait_for_seconds(20)

    adb_run_shell_command("monkey -p " + package_name + " -s 123 --throttle 500 -v 
2000 > monkey_log.txt")

    move_file("strace_output.txt", output_repository)
```

STATE DIAGRAM (SYSCALLS TRACING)

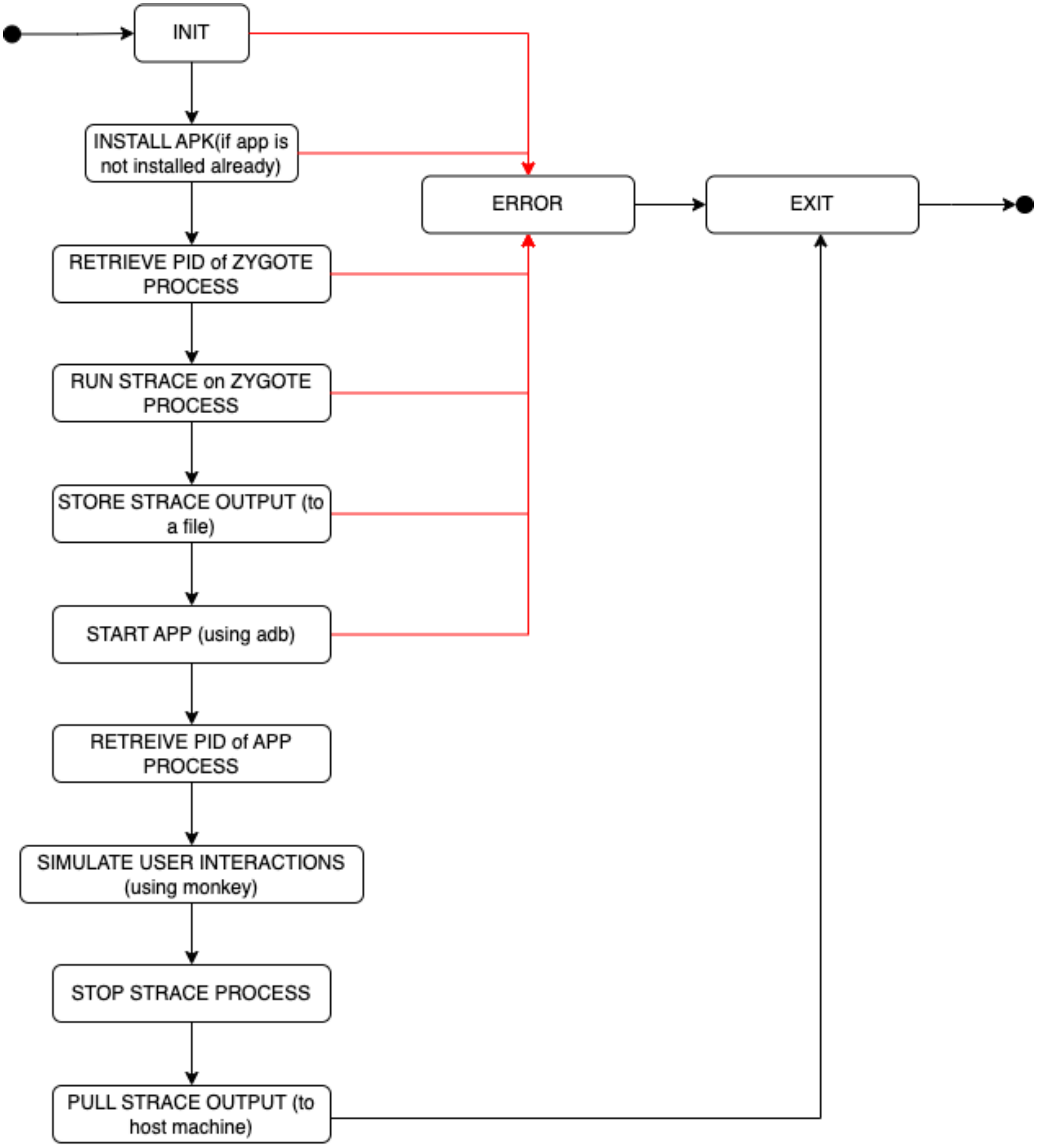

INIT
INSTALL APK(if app is not installed already)
ERROR
EXIT
RETRIEVE PID of ZYGOTE PROCESS
RUN STRACE on ZYGOTE PROCESS
STORE STRACE OUTPUT (to a file)
START APP (using adb)
RETREIVE PID of APP PROCESS
SIMULATE USER INTERACTIONS (using monkey)
STOP STRACE PROCESS
PULL STRACE OUTPUT (to host machine)

Fig: State Diagram of Syscalls Tracing

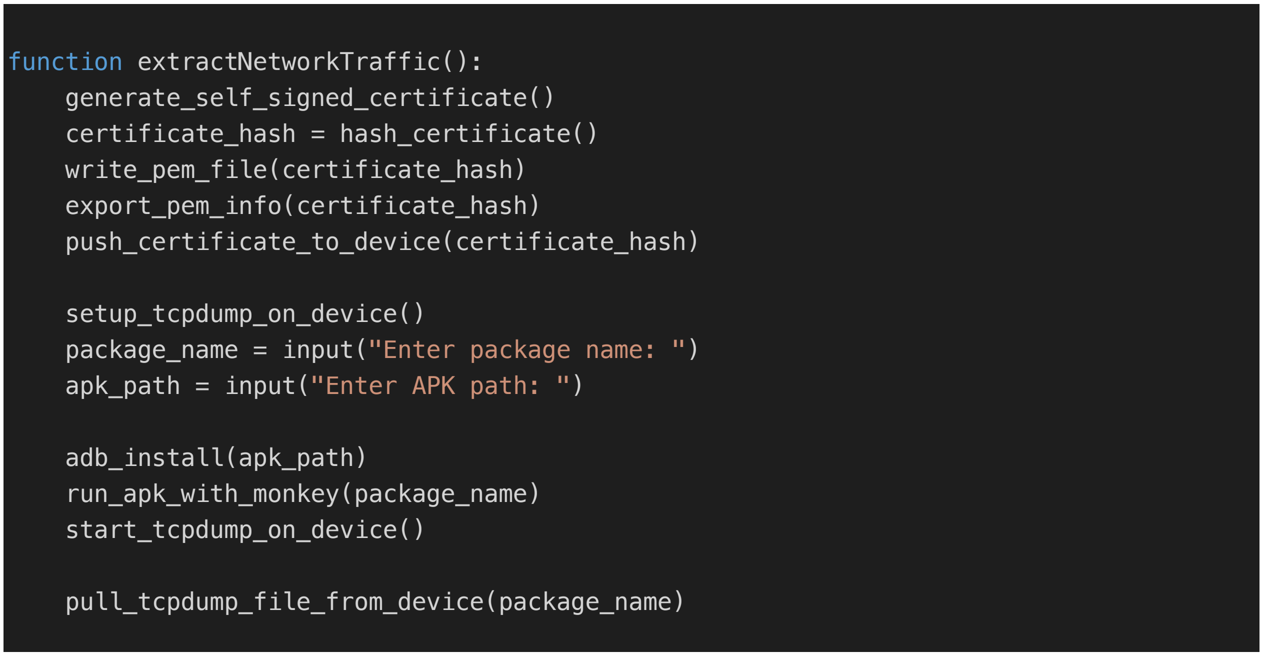
Extracting Network Calls:

This Python script automates the process of running an APK on an Android device, capturing the network traffic during the monkey enabled testing of the apk file.

It is achieved by performing the following tasks:

1. Generate a self-signed certificate.
2. Write the certificate to a PEM file and push it to the device.
3. Setup tcpdump on the device.
4. Install the APK on the device and run it with monkey.
5. Start tcpdump to capture network traffic.
6. Pull the tcpdump capture file from the device to your local machine.

Pseudo-code for extracting network traffic:



```
function extractNetworkTraffic():
    generate_self_signed_certificate()
    certificate_hash = hash_certificate()
    write_pem_file(certificate_hash)
    export_pem_info(certificate_hash)
    push_certificate_to_device(certificate_hash)

    setup_tcpdump_on_device()
    package_name = input("Enter package name: ")
    apk_path = input("Enter APK path: ")

    adb_install(apk_path)
    run_apk_with_monkey(package_name)
    start_tcpdump_on_device()

    pull_tcpdump_file_from_device(package_name)
```

STATE DIAGRAM (NETWORK TRAFFIC TRACING)

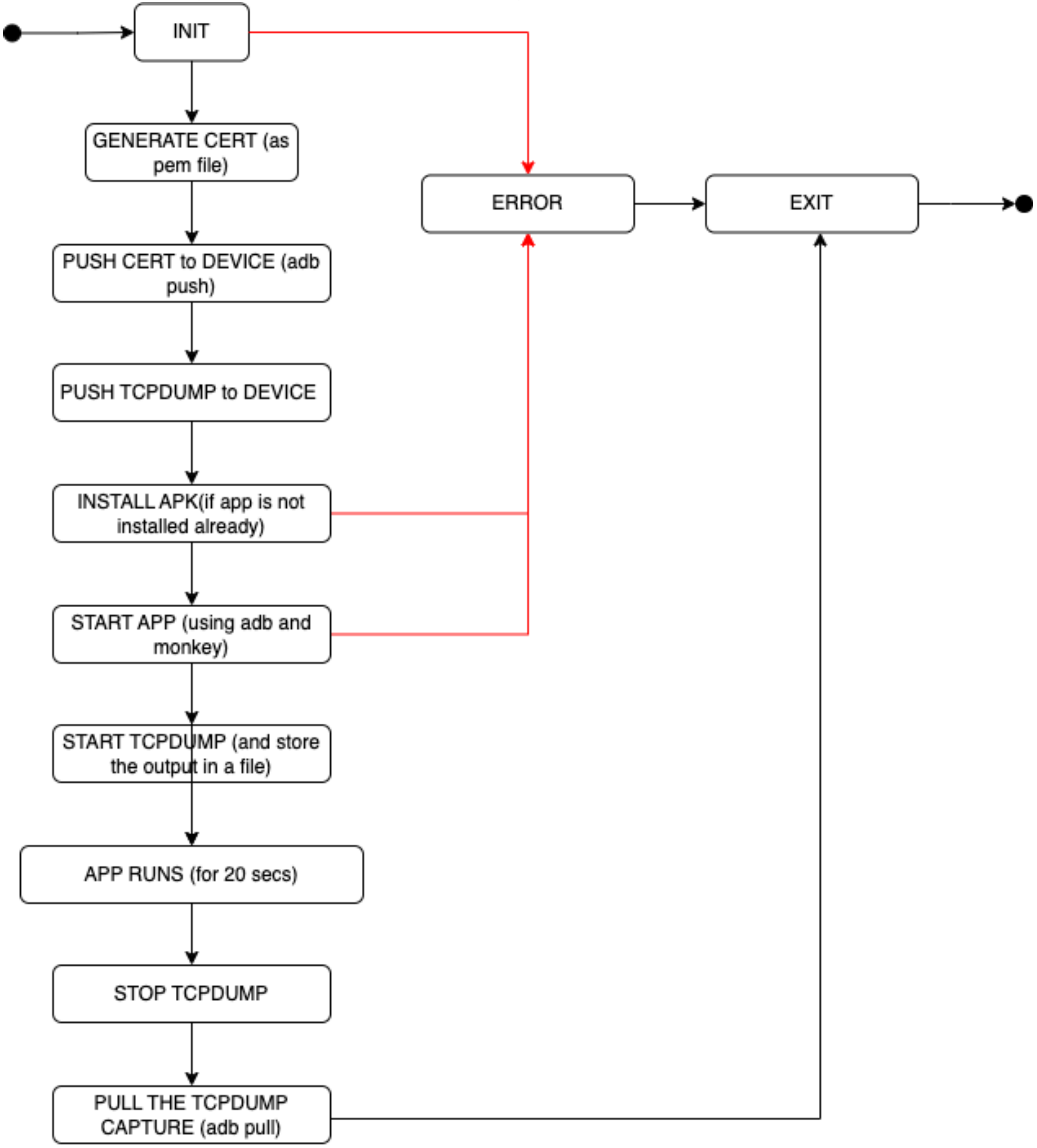
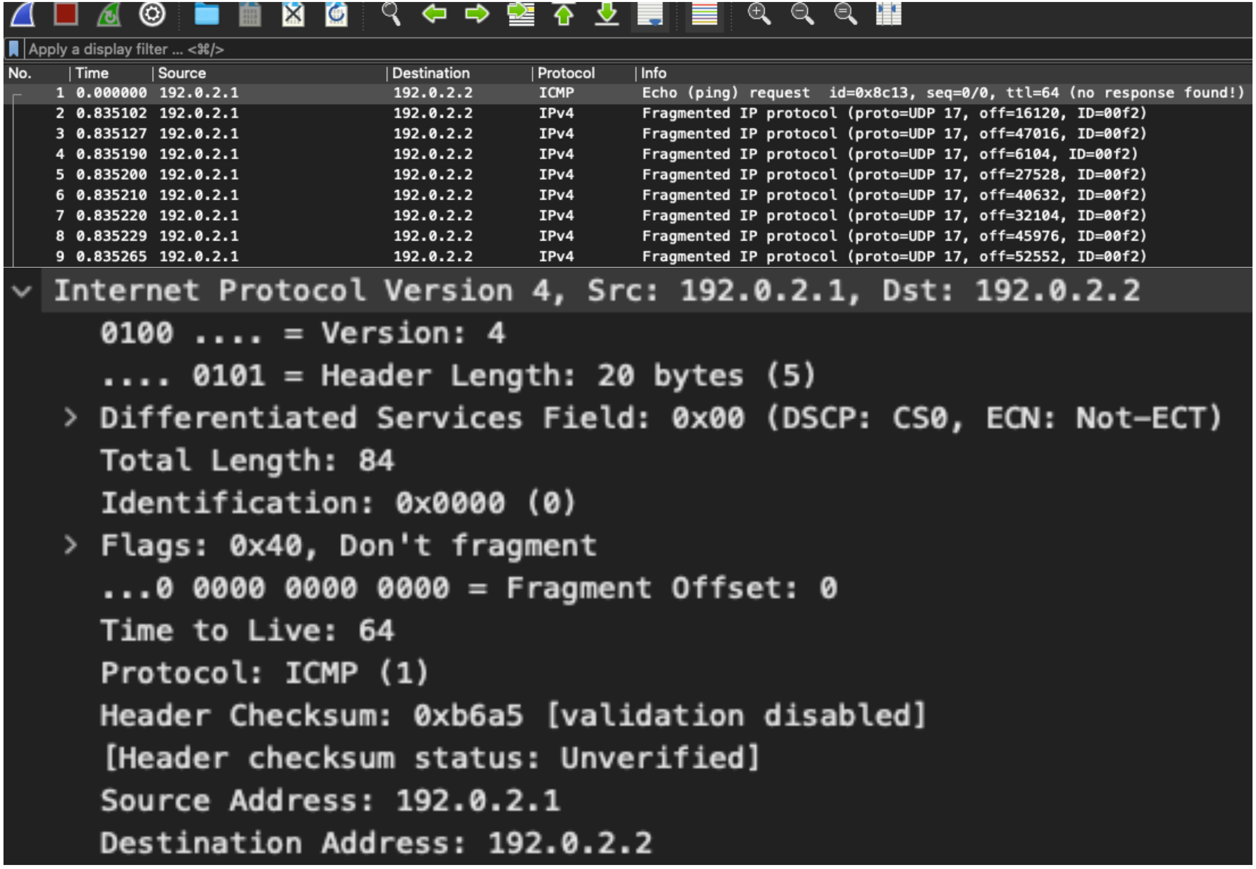

INIT
GENERATE CERT (as pem file)
PUSH CERT to DEVICE (adb push)
PUSH TCPDUMP to DEVICE
INSTALL APK(if app is not installed already)
START APP (using adb and monkey)
START TCPDUMP (and store the output in a file)
APP RUNS (for 20 secs)
STOP TCPDUMP
PULL THE TCPDUMP CAPTURE (adb pull)
ERROR
EXIT

Fig: State Diagram of Network traffic Tracing

Sample tcpdump dump pcap (file when opened in wireshark looks like):



```
No.   Time      Source          Destination   Protocol  Info
    1 0.000000  192.0.2.1       192.0.2.2     ICMP      Echo (ping) request  id=0x8c13, seq=0/0, ttl=64 (no response found!)
    2 0.835102  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=16120, ID=00f2)
    3 0.835127  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=47016, ID=00f2)
    4 0.835190  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=6104, ID=00f2)
    5 0.835200  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=27528, ID=00f2)
    6 0.835210  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=40632, ID=00f2)
    7 0.835220  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=32104, ID=00f2)
    8 0.835229  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=45976, ID=00f2)
    9 0.835265  192.0.2.1       192.0.2.2     IPv4      Fragmented IP protocol (proto=UDP 17, off=52552, ID=00f2)
```

```
v Internet Protocol Version 4, Src: 192.0.2.1, Dst: 192.0.2.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0x0000 (0)
  > Flags: 0x40, Don't fragment
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: ICMP (1)
    Header Checksum: 0xb6a5 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.0.2.1
    Destination Address: 192.0.2.2
```

- **Source IP and Port:** The IP address and port from which the communication originates. This could be the compromised machine.
- **Destination IP and Port:** The IP address and port to which the communication is directed. This might be an external server.
- **Protocol:** The protocol being used, such as TCP or UDP.
- **Payload:** The actual data being sent in the communication. This could include URLs, commands, or other information.
- **Connection Duration:** The duration of the communication session.
- **Packet Timing:** The timing and frequency of packets exchanged between the source and destination.

Based on the information extracted from the TCPdump file, we can identify connections to suspicious domains, unusual ports, or patterns of communication that do not align with normal network behavior. This could help with the investigation and analysis to determine if the app is malicious or not.

Suspicious indicators could be:

Suspicious Domains:

1. **Phishing Domains or Domains with typos:** Domains that closely mimic legitimate websites to deceive users into sharing sensitive information, such as login credentials or credit card details. Example: go0gle.com (instead of google.com)

2. **Newly Registered Domains:** Domains that have been registered recently and are not associated with reputable entities.

3. **Domains with Malicious Content:** Domains known to host malware, exploit kits, or other malicious content.

Unusual Ports:

1. **Non-Standard Ports:** Ports that are not commonly associated with the service or application being used or using very high port numbers that are not typically used for common services. Example: Using port 65432 for HTTP traffic or using port 9090 or 9000 for HTTP traffic instead of the standard port 80.

2. **Reserved Ports:** Ports that are typically reserved for specific services and not commonly used for regular communication. Example: Using port 25 (SMTP) for non-email traffic.

Unusual Flags (TCP Flags):

1. **FIN Scan:** Sending a TCP packet with only the FIN (Finish) flag set, without establishing a full connection. Often used for reconnaissance.

2. **Xmas Scan:** Sending TCP packets with the FIN, URG, and PUSH flags set. This can be used to identify open ports and potentially confuse firewalls.

3. **NULL Scan:** Sending TCP packets with no flags set. Similar to FIN scan, used for port scanning and reconnaissance.

4. **SYN Flood:** Sending a high volume of SYN (synchronize) packets to overwhelm a target system and cause a denial-of-service (DoS) condition.

5. **ACK Scan:** Sending TCP packets with only the ACK (acknowledgment) flag set to determine if a port is filtered or unfiltered.

6. **URG-ACK Scan:** Sending TCP packets with the URG and ACK flags set. Used to identify open ports on a target system.

7. **PUSH-ACK Scan:** Sending TCP packets with the PUSH and ACK flags set. Can be used to identify open ports and potentially detect poorly configured firewalls.

The integration of network call analysis and traffic assessment with Syscall monitoring forms a robust strategy for producing a predictive model to identify malicious apps. By combining insights from network behavior and system-level interactions, this fusion enhances the model's ability to detect and mitigate security threats in dynamic mobile app environments.

This approach strengthens defenses against evolving malware, providing a comprehensive strategy to safeguard user data and the digital landscape. This phase of the research underscores the significance of merging these analytical dimensions in securing digital ecosystems, which hasn’t been used by existing work in Dynamic analysis of android malware detection.

Extracting API calls:

Our initial approach involved the utilization of Frida[25] to be able to capture the api calls made by the app. However, our progress encountered a setback as a considerable amount of effort was directed towards rooting the physical device, a prerequisite for effective testing using Frida It's worth noting that prior work has explored the monitoring of APIs through Frida, exemplified by tools like PAPIMonitor (Python API Monitor)[26]. This is a Python script that utilizes Frida-tools to monitor the api calls made by an app with a caveat of being only able to run on the rooted device/emulator.

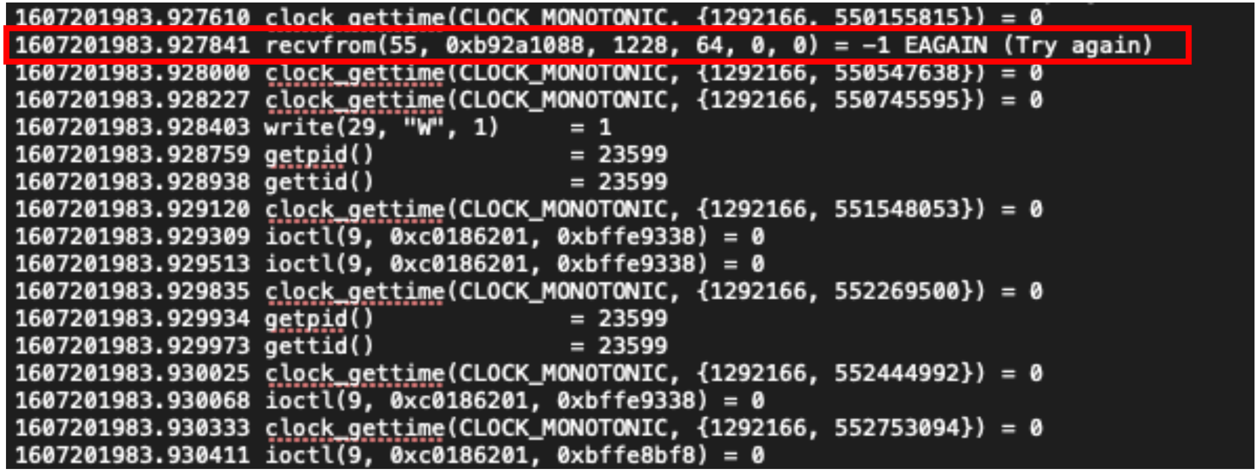
Another tool that is very popular in the community is **DroidMon[27]**, which was used by CuckooDroid[28] for their analysis on malware detection. DroidMon is based on Java. It works by hooking the zygote every time a package has been loaded. Droidmon opens the configuration file (location: /data/local/temp/hooks.conf), reads the list of API methods to monitor, hooks them dynamically every time a new application is opened, and reports all the monitoring information to logcat.

Phase 4: Collection of Raw data (apks) to be able to test the scripts against them and compare the results to what is produced by TwinDroid.

Source 1: AndroZoo
Source 2: Virus Share
Source 3: MalDroid
Source 4: APK Pure

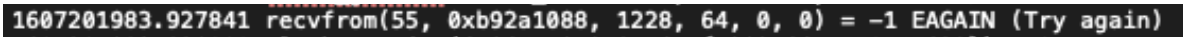
Tested ~70 apps from year 2020, and 2022 (APK Pure, Virus Share), their traces can be found on GH repo[10].

Example of Strace log
Here is a subset of strace logs:



```
1607201983.927610 clock_gettime(CLOCK_MONOTONIC, {1292166, 550155815}) = 0
1607201983.927841 recvfrom(55, 0xb92a1088, 1228, 64, 0, 0) = -1 EAGAIN (Try again)
1607201983.928000 clock_gettime(CLOCK_MONOTONIC, {1292166, 550547638}) = 0
1607201983.928227 clock_gettime(CLOCK_MONOTONIC, {1292166, 550745595}) = 0
1607201983.928403 write(29, "W", 1)      = 1
1607201983.928759 getpid()               = 23599
1607201983.928938 gettid()               = 23599
1607201983.929120 clock_gettime(CLOCK_MONOTONIC, {1292166, 551548053}) = 0
1607201983.929309 ioctl(9, 0xc0186201, 0xbffe9338) = 0
1607201983.929513 ioctl(9, 0xc0186201, 0xbffe9338) = 0
1607201983.929835 clock_gettime(CLOCK_MONOTONIC, {1292166, 552269500}) = 0
1607201983.929934 getpid()               = 23599
1607201983.929973 gettid()               = 23599
1607201983.930025 clock_gettime(CLOCK_MONOTONIC, {1292166, 552444992}) = 0
1607201983.930068 ioctl(9, 0xc0186201, 0xbffe9338) = 0
1607201983.930333 clock_gettime(CLOCK_MONOTONIC, {1292166, 552753094}) = 0
1607201983.930411 ioctl(9, 0xc0186201, 0xbffe8bf8) = 0
```

Dissecting this specific entry in the strace log:



```
1607201983.927841 recvfrom(55, 0xb92a1088, 1228, 64, 0, 0) = -1 EAGAIN (Try again)
```

- This line indicates a call to the **recvfrom** system call.
- The arguments are:
  - **55**: File descriptor for the socket.
  - **0xb9201088**: Destination buffer address.
  - **1228**: Maximum number of bytes to receive.
  - **64**: Flags (additional options for the call).
  - **0**: Source address (ignored in this case).
  - **0**: Source address length (ignored in this case).
- The return value is **-1**, indicating an error, and the error code is **EAGAIN** (Resource temporarily unavailable). This often happens when the socket is set to non-blocking mode, and no data is available for reading at the moment.

Phase 5: Taking inspiration from existing works for feature extraction from strace and network traffic logs

[10] traces

Tried to replicate feature extraction such as in **Evaluating Dynamic Analysis Features for Android Malware Categorization**[20]

The mentioned paper extracted a total of 141 dynamic characteristics including

- 23 memory features
- 105 API features
- 02 battery features
- 04 network features
- 01 process feature
- 06 logcat features

Result/Challenges:
The code from this paper isn't open-source and thus made it hard to reproduce their work. Tried reaching out to CopperDroid[21] (that's the tool they used to extract the features) but no reply from them either.

Response to the Challenges:
Extracted a few features on our own from the syscall traces generated by strace, that are:

1. **Syscall name**
2. **Syscall frequency**
3. **Parameters used for the syscall**
4. **Parameter value's frequency**

These extracted features can be found in the GH repo[11] as well.

These features were extracted using *extract-features-syscall* python script. The Python script reads a strace log file containing system call information, extracts relevant syscall entries, and then saves this data into a CSV file.

Why we chose to extract these features:

1. **Syscall Name:** This feature refers to the **specific system call** that was invoked during the execution of the Android app.

   System calls are fundamental operations that the app performs to interact with the underlying operating system. Certain system calls might be indicative of malicious behavior, such as trying to access sensitive data or performing unauthorized actions.

   Example: If we observe an app making repeated "$open$" or "$read$" system calls to access system files or sensitive user data without any legitimate reason**, it could indicate that the app is attempting to exfiltrate user information.

2. **Syscall Frequency:** This feature tracks how often a particular system call is invoked within the execution of the app.

[11] Extracted features

Unusual or high frequencies of certain system calls might indicate that the app is performing some action excessively, which could be an indication of malicious activity. **Example:** An app making an unusually high number of "$sendto$" or "$connect$" system calls might suggest that it's trying to establish unauthorized network connections or potentially sending out spam messages.

3. **Parameters Used for the Syscall:** This feature involves looking at the parameters passed to a particular system call.

   Different parameters provide context about what the app is trying to achieve with that call. Certain combinations of parameters might be indicative of malicious intent.

   Example: If an app is frequently calling the "$execve$" system call with parameters indicating that it's running commands or scripts from untrusted sources, it could potentially be involved in executing malicious code.

4. **Parameter Value's Frequency:** This feature involves analyzing the frequency of specific parameter values within a system call. Unusual or unexpected parameter values might suggest that the app is trying to manipulate the system in unintended ways.

   Example: If an app is making "$chmod$" system calls with the parameter values set to grant excessive permissions to sensitive files, it might be attempting to escalate its privileges and gain unauthorized access.

By examining these features collectively, we can establish patterns and anomalies that might indicate the presence of malicious behavior. It's important to note that while these features can provide valuable insights, they should be considered in conjunction with other analysis techniques, such as network traffic analysis, and behavioral analysis, to build a comprehensive understanding of an app's behavior.

**Determining whether the any system call is being used for a malicious intent is multi-fold:

1. **File Access Patterns:** Investigating the files that the app is trying to access using these calls. Look for patterns that might be indicative of malicious intent, such as accessing sensitive system files, personal user data, or configuration files.

2. **Unusual File Locations:** Paying attention to file paths that are typically not accessed by well-behaved apps. If the app is consistently trying to access files outside of its designated storage or data directories, it could be attempting to breach security boundaries.

3. **Frequency and Repetition:** High-frequency or repetitive use of the system calls to specific files might be suspicious. If an app is repeatedly accessing the same file rapidly,

it could indicate that it's attempting to exfiltrate data.

4. **Permission Violations:** If the app is making calls to files that it shouldn't have access to, based on its permissions, it could be trying to exploit vulnerabilities and access sensitive information.

5. **Network Behavior:** Correlating the system calls with the app's network behavior. If the app is accessing files and then sending them over the network, it might be exfiltrating data.

Examples of known malicious apps that illustrate the above-mentioned behaviour(s):

Examples of applications that exhibit such behaviors include **Psiphon**, **Boulders**, and **Currency Pro**. These apps have been found to contain a malware variant named "**daam**," which is designed to surreptitiously extract data from the user's mobile device[23]. The stolen data is then transmitted to a command-and-control server, which is operated by malicious actors. These applications display patterns of network behavior, as evidenced by the data transmission occurring over the network. Additionally, they consistently attempt to access files and directories beyond their designated locations.

Another illustrative case is the app named **Flashlight Plus**. This particular app requests an extensive array of permissions, indicating a potential attempt to exploit vulnerabilities and gain access to sensitive information. It also downloads a remote configuration by executing an HTTP request after one opens the app (exhibiting unusual Network behaviour). Additionally, it even registers a Firebase Cloud Messaging (FCM) listener to receive push messages to commit ad fraud on user's devices[24] (indicating permission violations and unusual file access locations).

The extraction of features is accomplished by following the below mentioned steps:

1. **Initialize:**
    - Set up android_syscalls list.
    - Prepare syscall_data and counter dictionaries.
2. **Read Log File:**
    - Open log_file for reading.
    - For each line in the log:
        - Check against android_syscalls.
        - Extract and track syscall names.
        - Populate syscall_data with syscall names and lines.
3. **Save to CSV:**
    - Define fieldnames for CSV columns.
    - Open csv_file for writing.
    - Create CSV writer with fieldnames.
    - Write header.
    - Write syscall data to CSV rows.

Pseudo-code for extracting network traffic:

```
function extractFeatures(strace_log_file):
    syscall_frequency = {}
    parameter_data = {}

    open strace_log_file for reading
    for each line in strace_log_file:
        syscall_name, parameters = extract_syscall_info(line)

        update_syscall_frequency(syscall_name, syscall_frequency)
        update_parameter_data(parameters, parameter_data)

    parameter_value_frequencies = {}
    for parameters in parameter_data:
        for parameter_value in parameter_data[parameters]:
            update_parameter_value_frequency(parameter_value,
parameter_value_frequencies)

    return syscall_frequency, parameter_data, parameter_value_frequencies

function extract_syscall_info(line):
    syscall_name = extract_syscall_name(line)
    parameters = extract_parameters(line)
    return syscall_name, parameters

function extract_syscall_name(line):
    return syscall name extracted from line

function extract_parameters(line):
    return parameters extracted from line

function update_syscall_frequency(syscall_name, syscall_frequency):
    increment syscall_frequency for syscall_name

function update_parameter_data(parameters, parameter_data):
    if parameters not in parameter_data:
        parameter_data[parameters] = []
    append parameters to parameter_data

function update_parameter_value_frequency(parameter_value,
parameter_value_frequencies):
    increment parameter_value_frequencies for parameter_value
```

STATE DIAGRAM (FEATURE EXTRACTION – SYSCALL TRACE)

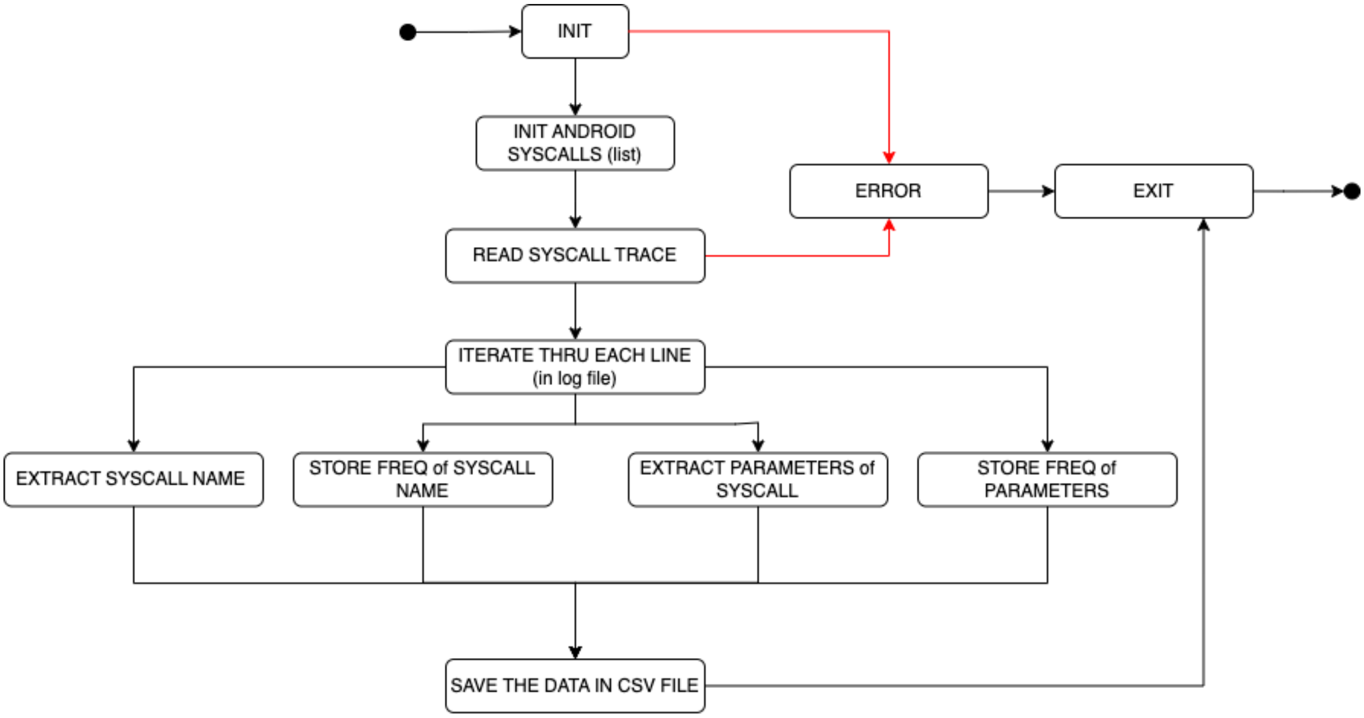

INIT
INIT ANDROID SYSCALLS (list)
ERROR
EXIT
READ SYSCALL TRACE
ITERATE THRU EACH LINE (in log file)
EXTRACT SYSCALL NAME
STORE FREQ of SYSCALL NAME
EXTRACT PARAMETERS of SYSCALL
STORE FREQ of PARAMETERS
SAVE THE DATA IN CSV FILE

Fig: State Diagram of Feature Extraction - SYSCALL TRACE

4. Future Work

1. **Enhancing User Interaction Simulation:** Instead of relying solely on the Android Monkey tool, which generates random interactions, consider employing more user-guided tools for a more accurate simulation of user behavior. This could involve using testing frameworks that allow you to script specific user flows, ensuring that the app's critical functionalities are thoroughly tested. By simulating actual user interactions, you can uncover usability issues, identify edge cases, and provide more realistic test scenarios.

2. **Utilizing an actual device rather than an emulator:** Most of the existing work in relation to dynamic analysis has been done on an emulator. Although emulator has its own pros, some malware apps have proven to detect being run on an emulator and change/ do not produce malicious data as a result of that. It is recommended to be able to find an Android phone that can be easily rooted and can be tested for our research.

3. **Feature Selection and Extraction from Logs:** The strace and tcpdump logs are rich sources of data that can provide valuable insights into an app's behavior and communication patterns. To enhance predictive modeling, you need to carefully select and extract relevant features from these logs. These features could include system call frequencies, network communication patterns, API calls made, and other relevant data points. By capturing this information, you create a more comprehensive representation of the app's runtime behavior, which can significantly improve the predictive model's accuracy.

4. **Producing a Comprehensive Vector:** The extraction of system calls, network calls, and API calls can be transformed into a feature vector that represents each app's behavior. Each element of this vector corresponds to a specific data point derived from the logs. By producing a structured and comprehensive vector, you create a meaningful input for the predictive model. This step essentially translates the raw log data into a format that the model can effectively analyze.

5. **Accuracy Computation:** After developing the predictive model, it's crucial to assess its accuracy. Accuracy measures how well the model's predictions align with actual outcomes. This involves evaluating the model's performance using appropriate evaluation metrics such as precision, recall, F1-score, or area under the ROC curve (AUC-ROC). These metrics provide a quantitative assessment of the model's ability to correctly classify instances.

5. References

[1] Published by Statista Research Department, and Aug 9. “Global Mobile OS Market Share 2023.” *Statista*, 9 Aug. 2023, www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[2] Kaspersky. “Android Mobile Security Threats.” *Www.Kaspersky.Com*, 19 Apr. 2023, www.kaspersky.com/resource-center/threats/mobile.

[3] Kovacs, ByEduard. “New Samsung Message Guard Protects Mobile Devices against Zero-Click Exploits.” *SecurityWeek*, 20 Feb. 2023, www.securityweek.com/new-samsung-message-guard-protects-mobile-devices-against-zero-click-exploits/.

[4] Fedler, Rafael, Julian Schütte, and Marcel Kulicke. "On the effectiveness of malware protection on android." *Fraunhofer AISEC* 45 (2013): 53.

[5] Razagallah, Asma, Raphaël Khoury, and Jean-Baptiste Poulet. "TwinDroid: a dataset of Android app system call traces and trace generation pipeline." *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022. https://github.com/RaphaelKhoury/automated-apk-tracing

[6] Cai, Haipeng. "Assessing and improving malware detection sustainability through app evolution studies." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29.2 (2020): 1-28. https://bitbucket.org/haipeng_cai/droidspan/src/master/

[7] https://github.com/maryam-tanha/DynamicAndroidMalwareAnalysis

[8] Diagrams

[9] SwiftHand, SwiftHand2, Calabash Android, Appium

[10] traces

[11] Extracted features

[12] https://developer.android.com/tools/adb

[13] https://developer.android.com/tools/aapt2

[14] https://developer.android.com/studio/test/other-testing-tools/monkey

[15] https://man7.org/linux/man-pages/man1/strace.1.html

[16] https://www.tcpdump.org/

[17] https://frida.re/

[18] https://www.openssl.org/

[19] https://www.genymotion.com/

[20] S. Khalid and F. B. Hussain, "Evaluating Dynamic Analysis Features for Android Malware Categorization," 2022 International Wireless Communications and Mobile Computing (IWCMC), Dubrovnik, Croatia, 2022, pp. 401-406, doi: 10.1109/IWCMC55113.2022.9824225.

[21] Tam, Kimberly, et al. "Copperdroid: Automatic reconstruction of android malware behaviors." *Ndss*. 2015.

[22] Adrienne Porter Felt , Matthew Finifter , Erika Chin , Steve Hanna , David Wagner, “A survey of mobile malware in the wild”, Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, October 17-17, 2011, Chicago, Illinois, USA

[23] Spadafora, Anthony. “Daam Android Malware Can Hold Your Phone Hostage - What You Need to Know.” *Tom’s Guide*, 27 Apr. 2023, www.tomsguide.com/news/daam-android-malware-can-hold-your-phone-hostage-what-you-need-to-know#:~:text=While%20Psiphon%20is%20a%20VPN,infected%20with%20the%20Daam%20malware.

[24] Spadafora, Anthony. “These 16 Malicious Android Apps Have over 20 Million Downloads - Delete Them Now.” *Tom’s Guide*, 22 Oct. 2022, www.tomsguide.com/news/these-16-malicious-android-apps-have-over-20-million-downloads-delete-them-now.

[25] https://frida.re/

[26] https://github.com/Dado1513/PAPIMonitor

[27] https://github.com/idanr1986/droidmon

[28] https://github.com/idanr1986/cuckoo-droid

FOOTNOTE:
All the diagrams used in this report or earlier during the entire research term can be found here: Diagrams