System Calls Monitoring in Android: An Approach to Detect Debuggers, Anomalies and Privacy Issues

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Master's Degree programme in Computer Science
Final Thesis

Academic Year 2021-2022

Graduand
Matteo De Giorgi
Matriculation Number: 872029

Supervisor
Prof. Paolo Falcarin

Abstract

The proposed thesis explores monitoring system calls in Android environments to detect the presence of debuggers, identify anomalies that can be indicators of security issues, and observe how user-sensitive data is handled. System calls are fundamental for every application since they are the mandatory gateway to request an action from the operating system; therefore, accessing any resource implies performing one.
To achieve these goals, a system call capturing and analyzing tool named *Ptracer* has been developed. It places itself between an application and kernel to intercept every interaction among them and gather information like the stack backtrace and used parameters for each observed system call.
Moreover, the captured information can be represented in a model based on a Non-deterministic Finite state Automaton (NFA) and refined during multiple learning iterations, effectively linking all the observed kernel interactions by a causal relationship. Such a model describes what is considered a “normal” application behaviour and will be used to detect anomalies by enforcing it during future application executions.
The collected information will be extremely useful in detecting whether an external actor is trying to debug, tamper or breach the application since such attempts would alter its normal behaviour, execution speed, or pace. The final results will show how system calls interception is a rich source of information that can be used to protect the application from various attacks. Furthermore, by analyzing what actions are requested to the kernel, it is possible to determine what sensitive data the application requests and how often, with the goal of identifying privacy issues.
The proposed future developments aim to reduce *Ptracer*'s analysis overhead, actively protect user privacy, and provide new and more sophisticated techniques for detecting MATE attacks and anomalies. These future goals will be achieved by improving the analysis quality to reach a deeper insight into the application and expanding the behavioural model by including different data types to counter a wider variety of attacks (e.g., DoS attacks). Moreover, new interception technologies like eBPF will be considered and discussed.

Keywords Syscall Monitoring, Android Security, Debugger Detection, Anomaly Detection, Android Privacy

Contents

Contents iii

List of Figures v

1 Introduction 1
1.1 MATE Attacks . . . 1
1.2 Intrusion Detection and Prevention . . . 2
1.3 Privacy . . . 3

2 Related work 4
2.1 MATE Attacks . . . 4
2.2 Anomaly Detection . . . 5
2.3 Privacy monitoring . . . 6

3 Background 8
3.1 Linux System Calls Tracing . . . 8
3.2 Executable Structure . . . 9
3.3 Android applications . . . 9
3.4 Android Binder IPC . . . 9

4 Solution 11
4.1 Architecture . . . 11
4.2 System calls monitoring . . . 14
4.3 Stack trace extraction . . . 16
4.4 NFA Model . . . 21
4.5 System call decoders . . . 27
4.5.1 File Decoder . . . 30
4.5.2 Socket Decoder . . . 32
4.5.3 Ptrace Decoder . . . 36
4.5.4 Binder IPC Decoder . . . 37
4.5.5 Execve Decoder . . . 47

5 Validation 49
5.1 Use case . . . 49
5.2 Anomaly detection . . . 53
5.2.1 Attach to the application . . . 54
5.2.2 Learn the model . . . 57
5.2.3 Enforcement phase . . . 62
5.3 Debugger detection . . . 68
5.3.1 Architecture . . . 69

5.3.2 Debugger as an anomaly 70
5.3.3 Time checks 73
5.4 Privacy issues 75

6 Future Developments **80**
6.1 eBPF 80
6.2 System Call Decoders 80
6.3 Attach to Zygote 81
6.4 Improve the tracing mechanism 81
6.5 Model improvements 82
6.6 Debugger detection evolutions 82

7 Conclusions **84**

Appendices **86**

A Build and installation **87**
A.1 Usage 87
A.2 Dependencies 89
A.3 Build 89
A.4 Debug 90

Bibliography **91**

List of Figures

4.1 Internal architecture of Ptracer in an Android environment 12
4.2 Positioning of Ptracer in an Android environment 15
4.3 Partial output of tracing `ls` in Android without capturing the stack trace 16
4.4 Partial output from *Ptracer* of a system call coming from an Android AudioRecorder including its stack trace 18
4.5 Diagram showing where the various information extracted by *Ptracer* point in the example program EvenOdd 20
4.6 Source code of the program EvenOdd with the resulting NFA 21
4.7 Redacted example of system calls and stack traces mapped by the *Mapper* function for EvenOdd 22
4.8 Interface implemented by every system call decoder 28
4.9 Architecture of the System Call Decoders 29
4.10 Final report generated by the Open Decoder 32
4.11 Final report generated by the Socket Decoder on a generic Linux system 35
4.12 Final report generated by the Socket Decoder on an Android system 35
4.13 Final report generated by the Ptrace Decoder 36
4.14 Representation of the Read/Write buffer structure extracted by the Android Binder Decoder 42
4.15 Snippet of the services exposed via the Binder with their interface declaration 43
4.16 Snippet of the `ITelephony` interface definition containing the method that will be called 43
4.17 Binder Decoder output showing the application request for a handle to the `phone` service 44
4.18 Snippet of the `IServiceManager` interface definition containing the method that will be called 44
4.19 Binder Decoder output showing the claim of a handle and a request to free a buffer 45
4.20 Binder Decoder output showing the application acquiring the interface descriptor of the `phone` service 46
4.21 Binder Decoder output showing the remote method call of the `dial` method 46
4.22 Binder Decoder output showing the release of a handle and a request to free a buffer 47
4.23 Execve Decoder output showing the shell executing a command . . 48

5.1 Main screen of the test application used for the validation 50

5.2 Source code snippet from *AudioRecorder* showing how the recording is started . . . 51
5.3 Source code of the `Configurations` class . . . 52
5.4 Source code snippet from *AudioRecorder* showing how the configurations file is read . . . 52
5.5 Process tree of *AudioRecorder* . . . 55
5.6 Partial representation of the NFA model obtained after the first learning iteration on *AudioRecorder* . . . 58
5.7 Plots the NFA model status during the various learning iterations . 61
5.8 Unknown State Anomaly found while restoring the application . . . 63
5.9 Unknown Transition Anomaly found while reopening the application 64
5.10 Anomaly found when trying to exploit the unsafe deserialization vulnerability . . . 65
5.11 Execve state triggered when the unsafe deserialization exploit is used 67
5.12 Architecture used for detecting debuggers . . . 69
5.13 Anomaly detected in the application started by the Java debugger . 71
5.14 Anomaly detected in the application after the Java debugger attached 72
5.15 Plots showing the different execution times when a debugger is attached . . . 74
5.16 Binder Decoder snippet showing the request to record on a specific file descriptor . . . 76
5.17 Snippet of the File Decoder final report after the application has started recording . . . 77
5.18 List of interfaces used by *Instagram* during the test . . . 77
5.19 RPC performed by *Instagram* on the Android location service . . . 78
5.20 Code snippet from the AIDL interface `ILocationManager` . . . 78
5.21 RPC performed by *Instagram* on a *Facebook* AIDL service . . . 79

Chapter 1

Introduction

A System Call is the main way used by user-space processes to request a service from the operating system and it is part of the kernel interface. They are performed temporarily transferring control to the kernel and are needed to perform any sort of process creation and management, file access, file system management, I/O, networking, and more. Without performing a system call a process can only perform calculations and operate on its own volatile memory, even its termination would require one. Linux systems offer various ways to gain control over system calls performed by any application, one of the most common ones is using `ptrace`, which effectively allows a process in user space to control another one. Taking control of every communication between an application and the kernel is an effective way to have a complete view of its behavior since every potentially malicious action must involve a system call. Moreover, altering system calls it is possible to This thesis focuses on what can be achieved by monitoring and potentially altering system calls performed by an application running on the Android operating system.

Android has become the most popular mobile operating system worldwide, with millions of users and developers relying on its open-source platform for their daily activities. This implies that a wide amount of people are very often in the proximity of an Android system capable of capturing a variety of information, and every software able to run on this platform is bound to have a vast potential user base. Inevitably, its popularity has also made Android a target for malicious actors, who are interested in gaining sensitive information, control over the whole device, and software piracy. Achieving any of these goals can imply a strategic and economical advantage for attackers, especially in all those cases where information secrecy is critical or it is important to protect the program's internal mechanics and data flow.

This thesis proposes an approach to monitoring application behavior based on System Calls inspection, analysis, and correlation with the end goal of detecting reverse engineering attempts, security intrusions, and privacy issues.

1.1 MATE Attacks

Man-At-The-End (MATE) attacks occur in any setting where an adversary has physical access to a device and aims to compromise it. They are particularly

hard to detect and prevent for various reasons: the attacker has limitless and authorized access to the target; all major protections stand up to a determined attacker only for a certain period of time; the attacker is human and, therefore, utilizes motivation, creativity, and ingenuity. Hence there are various software protection primitives that aim to address MATE attacks, but none of them is expected to hold off an attacker for an indefinite period of time since with enough motivation, time, and money all protections will eventually fall.

Reverse engineering is often one of the first steps in a MATE attack, it consists in the act of dismantling a previously made device, process, system, or piece of software to understand how it accomplishes a task. Since it can be a means to find vulnerabilities and pirating software, it is often prohibited by the software terms of service, and multiple technical measures are set in place to make it harder.

To reverse engineer an application one of the most useful too is a Debugger, which can be used to analyze the behavior of an application, it can help developers detect and solve bugs running the target program under controlled conditions allowing them to track its execution and identify malfunctioning code. At the same time debuggers can give attackers valuable insight into what algorithms and data are used, especially when paired with decompilers, they can provide a combined view of what algorithms are used, what data is flowing, and a human-readable version of the running instructions. Moreover, Android applications are often written in Java, given the wide support for it offered by the platform, which is an interpreted language that includes a lot of metadata in its bytecode and makes it easy to retrieve the original source code.

In literature, some of the techniques used to hinder MATE attacks are often categorized in the following categories: Obfuscation, which allows transforming the program into a new one with the same functionality but harder to analyze for an attacker; Tamper-Proofing, which consists in transforming the code by adding self-checks aiming to detect if the intended program behavior has been altered and react consequently; Watermarking, which allows unauthorized copies to be traced by embedding unique identifiers in a piece of software [8]. The work done in this thesis wants to propose a new technique to address MATE attacks detecting the presence of a debugger, this will be done by modeling the acceptable time between one system call and another and the acceptable sequences of calls. Hence, allowing identifying debuggers, since attaching one or setting a break-point will heavily delay the modeled times, and modifying the code will likely alter the sequence of allowed invocations.

1.2 Intrusion Detection and Prevention

An Intrusion Detection System (IDS) is a monitoring system that aims to detect malicious activities or policy violations and raise alerts when found. To do that a variety of data can be used, some of the most common IDS use network traffic, since it is one of the main vectors to attack a system, but there are also IDS based on events that occur within a host at various levels. The type of data used also offers a way to categorize them, in this thesis a Host-based Intrusion Detection System will be proposed (HIDS), since the acquired data come from the operating system.

Another way of categorizing IDS is based on their detection method, hence how they decide what behavior is malicious or allowed. Their detection policies are often subdivided into categories based on how their models are built: A specification-based detection scheme is used when patterns of allowed activities are known, a Misuse detection scheme base its decision on a known list of malicious activities, and an Anomaly detection scheme is used when patterns of normal activities are known. This thesis focuses on HIDS based on Anomaly detection, since System Calls will be observed during a learning period, a model of normal activities will be built and eventually used to enforce it.

This approach can also be used to counter MATE attacks preventing code tempering since such an attacker is going to be interested in manipulating the strategic part of code in order to gain various advantages (e.g., bypassing license checks or cheating in a game). Once learned what is the normal sequence of System Calls and their stack trace, it is possible to enforce it by detecting behaviors that cannot happen.

1.3 Privacy

Android devices often have access to a multitude of sensors (GPS, microphone, camera, and more), which, combined with their high level of connectedness to the Internet and other devices, exponentially increase the risk of leaking sensitive information either to malicious actors or companies (e.g., for advertisement profiling). Recent versions of the Android operating system provide a granular permission system that forces every application to explicitly request authorization to use every sensor. Despite this, once granted, it is not possible to see on how often a permission is used or what happens to the captured information. For example, once permission to access the microphone is granted the user has no way to know if the application is using this privilege just when expected or if it is constantly listening and sending data to a remote server.

This thesis wants to propose a way of monitoring what an Android application is doing with the granted permissions, what files are opened, and identify where the data flows. Eventually, trying to correlate the exfiltrated data with the information acquired from the operating system.

Since every acquisition of sensitive information needs to happen to invoke some System Calls then monitoring them allows having a view of what data has been requested and provided to the application. The project developed as part of this thesis is also able to provide an understanding of what actions have been attempted by observing the System Calls and providing a final summary. Moreover, it also allows modification of the data provided to the application by the operating system in order to preserve user privacy. For instance, when the system call `open` is observed then the application is trying to get access to a file, reading its path from the parameters of the call and looking for the returned file descriptor in the subsequent `read` and `write` will allow having a higher level view on what application is trying to achieve.

In the next chapters, it will be analyzed how Android applications can request permission to access a sensor, how data can be acquired from it via the Hardware Abstraction Layers, and how these actions translate into System Calls.

Chapter 2

Related work

This chapter wants to provide an overview of what is the current state of the art of the three main problems addressed in this thesis: Debugger detection, Anomaly detection, and Privacy monitoring.
All three areas have seen an increasing interest in the last years, especially in the context of Android applications, which is going to be the scope of this work, together with the application of System Call analysis.

2.1 MATE Attacks

Software protection has been an intrinsic problem of software engineering since commercial software appeared, which made it crucial to mitigate MATE attacks aiming to protect the integrity of data and code of applications running on untrusted devices. Hence, over time it has become more evident that it is necessary to develop solutions to make reverse engineering, piracy and tampering harder for an attacker. Some of the solutions developed try various measures to hide data and code [12, 4], detect and prevent tampering, perform remote attestation of devices [32] and code renewability [2].

There are various implementations of the techniques described above, two of the most complete examples are the ASPIRE Project [9], and Tigress [7]. The first targets mobile devices and aims to establish a trustworthy software execution on untrusted platforms; it does that by proposing multiple lines of defence into a plugin-based software protection toolchain [11].
On the other hand, Tigress is also able to apply multiple layers of defence and it is described as a source-to-source transformer for the C language that supports many defences against both static and dynamic reverse engineering and de-virtualization attacks.

In the specific area of debugger detection and prevention, the ASPIRE Project offers an implementation of self-debugging [1]. This protection tightly couples a custom debugger to the application to protect and migrates code fragments to the debugger, effectively making reverse engineering harder and preventing other debuggers from attaching to the application since the only available seat will be taken.

The OWASP foundation offers an extensive section in their Mobile Application

Security Testing Guide (MASTG) [5] where various Android Anti-Reversing Defenses are described.
The guide proposes various solutions that can be composed together, forming a more extensive multi-layer approach to preventing and detecting reverse engineering attempts. These solutions involve detecting if an Android device has been rooted or does not have an official ROM, preventing and/or detecting debugging in various ways, identifying tampering checking executable files and much more.
One of the techniques proposed is using timer checks, and it is based on the fact that a debugger slows down process execution, then measuring how much time it takes to execute a specific code section would give an indication of a potentially attached debugger.

Nowadays, mobile devices are often embedded with many sensors that can provide helpful information regarding the environment surrounding the platform; they can also be used to determine if the application is running in an emulator [30, 28].
This technique involves gathering data from as many sensors as possible to identify anomalies that the running platform is not a real user device. For example, if the motion sensor never detects any movement, not even when the user touches the screen, then this can be considered a first hint that the platform might not be legit.
When multiple hints are raised, then a server can consider an execution platform as untrusted and take the appropriate measures.

The debugging detection approach proposed in this thesis is going to extend time checks to the whole program, offering a way to apply it to all the system calls generated by an application.

2.2 Anomaly Detection

Starting from the mid-90s, it is possible to see an interest in developing intrusion detection systems that base their choices on System Call analysis; for example, [17] proposes an n-gram model to validate small sequences of calls.
System Calls were soon identified as a valuable indicator to respond to the question: can the application behaviour be considered normal?
Various detection techniques based on System Calls analysis have emerged in the following years, some determining the normal behaviour of the program through static analysis [33], others using dynamic analysis [24] or a combination of both [27] to leverage on the advantages of both approaches.

There has been an evolution also on the models used to capture the expected behaviour of an application; automaton transition verification was first described in [24], then formalized first as a Finite State Automaton (FSA) in [29], and then as a Non-Deterministic Finite State Automaton (NFA) in [33], eventually it has been shown that the call stack provides a valuable contribution in detecting anomalies, and the model has been further improved using Push Down Automata (PDA) leveraging on their stack to maintain the function call context [33].
All the previously mentioned models are based on static analysis, even though they also have applications in black-box contexts. They have evolved in the most recent state-of-the-art models such as Dyck [21], VPStatic [16], and the Inlined Automaton Model (IAM) [23], in the attempt to reduce the overhead of the PDA approach.

Using a black-box or grey-box approach, other more advanced paradigms have been developed to detect anomalies. Some notable examples are: VtPaths which utilize return address information extracted from the call stack to build virtual paths [15], Execution Graphs which provide a grey-box approach that accepts only system call sequences consistent with the program control flow graph [19], hidden Markov models where the hidden stochastic process are the aggregated tasks performed by the process (e.g., reading a file) and is observed by the emitted system calls [20], and its improved STILO model [35].

Very different are the approaches based on various data mining [26] and neural network models, which are recently evolving.
Other contributions approach this problem from a different level, e.g., from the hardware point of view, hence proposing an even stricter semantic checking not only system calls but also jumps [6].
This contribution proposes a further analysis layer that can be placed on top of other existing approaches to learn properties on system call parameters [38].

In the context of Android applications, similar models have been applied to System Call monitoring to detect anomalies in the form of malware [13, 36, 3], where classification algorithms have been used to discern benign and malicious behaviours.
Other contributions aim to provide a different level of insight on the actions performed by an application [40, 31, 39] which can help analyse malware.
At the moment of writing this thesis, a contribution targeting generic anomalies in Android systems is unknown.

This thesis would like to provide a practical approach that specifically targets Android applications and aims to construct a behavioural model based on NFA and Stack traces similar to VtPaths [15] and Execution Graphs [19].

2.3 Privacy monitoring

Given the wide adoption of Android and the level of information sensitivity that is often handled in mobile devices (e.g., banking apps, sensors, etc.), there has been a rising interest and number of contributions in the area of detecting Privacy issues in understanding how applications use the granted permissions.

Since Android applications are often executed in an interpreter VM, hence based on languages that are able to produce bytecode targeting that platform (e.g., Java and Kotlin), a first attempt has been made using static analysis to identify code portions that acquire sensitive data or communicate with the Internet. This solution is adequate but not complete since the application can also dynamically at run-time download and execute code from a remote server.

Other approaches involve using dynamic analysis at various levels to gain an insight into the application behaviour at runtime; TaintDroid [14] uses a modified version of the interpreter VM to perform a taint analysis on the acquired data, hence aiming to trace the full path of sensitive information and have a clear picture of what data is leaked. One of its limitations is that it is able to monitor only the interpreted instructions and does not have any insight into native code.

Projects like CopperDroid and DroidScope [31, 25] offers a virtualization-based malware analysis platform and a view from the point of view of the QEMU Android

emulator, parsing system call invocations to detect what sensitive data has been acquired. This approach is very powerful for malware and has goals similar to this thesis, but it does not target real devices, contrary to the approach proposed in this work.

Aurasium [37] offers an approach at a different level since it repacks applications inserting user-level sandboxing and policy enforcing code to be able to watch for security and privacy violations. It effectively places itself between the code and the native libraries used. This technique is very effective in determining what and how sensitive data is handled but presents some limitations: the presence of the new instrumentation code can be easily spotted, using native code, it would be possible to circumvent Aurasium, and the used sandbox is prone to "time of check/time of use" (TOCTTOU) race conditions.

Other tools use a combination of static and dynamic analysis and leverage on the Linux process tracing interface `ptrace`; for example, DroidTrace [40] has been designed for studying malware and uses static analysis to identify code sections that dynamically load new code, and dynamic analysis to monitor all the application behaviours. Despite the fact that its original goal is not identifying privacy issues, its analysis also provides valuable information in this regard.
Another relevant example is ProfileDroid [34]; it aims to be a system for monitoring and profiling Android applications. It uses static analysis to identify what permissions are requested by the application and if Intents are used for accessing resources indirectly through other apps. The tool `strace` is used to obtain a view of the flow of system calls, and `tcpdump` is used for inspecting the network layer. The final analysis is not thorough as it does not perform a deep inspection of what Inter-Process Communications (IPC) is performed.

This thesis would like to propose a new system that targets real Android devices, based on dynamic analysis via `ptrace`, and is able to deeply inspect system calls that might impact the user's privacy, e.g., IPC calls.

Chapter 3

Background

The approach proposed in this thesis relies on `ptrace` to halt the traced process every time a System Call occurs and be notified in order to have the chance to extract the stack trace. Moreover, it is focused on Android, which has its particularities and complex system design.
This implies that there are multiple aspects related to the structure of executable files, their linking to libraries and how Android applications are executed, which need to be considered. This chapter aims to give a high-level overview of the relevant aspects of these topics.

3.1 Linux System Calls Tracing

The `ptrace` Linux kernel process tracing interface (in place since kernel 1.0) provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee") and examine and change the tracee's memory and registers. In this context, the tracee will always be referred to as a generic process, which could also be a thread, also known as a lightweight process.

This interface can be used via the system call `ptrace`, which allows one first to specify the identifier of the process to trace and then specify how it should be traced. The identifier is normally known as the Process ID (PID), but it also accepts a Thread ID, which in Linux terms is often referred to as SPID. If the PID of a thread group leader is selected, this does not mean that also the system calls of all the threads in its group will be received.
Moreover, it is important to notice that once a tracer is attached to a tracee, no other tracers for the same process (or thread) are admitted.

In an Android environment, the usage of `ptracer` is further restricted in such a way that only the parent of a process can trace it. Therefore, making it impossible to attach to an application since they are always executed by the same parent (as will be described later).
A way around this limitation is running the tracer process as root, granting it the possibility to attach to any process.

3.2 Executable Structure

When an executable requires to use of a library, it can either embed it and hence be statically linked to it or leverage on the shared libraries present in the system. At runtime, the linker will ensure all the right libraries are loaded before starting the application.
One very common protection against buffer overflow and ROP attacks is Address space layout randomization (ASLR), which randomizes the memory structure of a program every time it is run. Hence, each execution will have libraries, stack and heap in different memory positions.

Despite this being a very strong mitigation against attacks, it also poses a problem while acquiring stack traces, as will be discussed later. To circumvent this issue, instead of leveraging on Program Counter addresses, the function name will be extracted.

3.3 Android applications

Every Android system has a Zygote process, which preloads all the system resources and classes used by the Android Framework, it is launched at the system startup and is the parent of every running application.
To speed up the execution of new applications, the Zygote process maintains a pool of its clones that are ready to specialize into an application once needed. This pool is called the "Blastula Pool".

The Android runtime (ART) is the managed runtime used by Android applications. ART, as the runtime, executes the Dalvik Executable format and Dex bytecode specification.
Dalvik is a discontinued process virtual machine (VM) in the Android operating system that executes applications written for Android (Dalvik bytecode format is still used as a distribution format, but no longer at runtime in newer Android versions.)

The particularity about Android applications is that they do not have a single entry-point, but being heavily event-driven they can have multiple, as it will be seen in the next sections.

3.4 Android Binder IPC

Processes in Android have separate address spaces, and a process cannot directly access another process's memory. Therefore, they need a way to communicate among themselves, and this is where the Android Binder IPC comes to the rescue. The Binder is an inter-process communication (IPC) mechanism heavily used in every Android application since, without it, no system interaction would be possible, even receiving the event for a touch on the screen requires interacting with it.

It allows waiting for events, e.g., the completion of a task or just a user input to come.
Moreover, it is one of the ways to enforce Android permission since every time an

application needs to use a service exposed by the system, e.g., the microphone, it will need to perform a Remote Procedure Call to the service exposing the methods to read from it, methods who will check if the requester has enough permission to use the microphone.

It allows to share memory transparently and to use reference counting for objects, this enables to efficiently exchange data between applications and system services since, for example, the latter could directly write on a file descriptor provided by the application.

The Binder also supports exchanging messages between applications or services, this is done by encapsulating them in Parcels, which are containers that support being sent through the Binder.

Effectively, this component can be considered the “heart” of Android since it is extremely important for every application and enables all the system components to communicate efficiently.

Chapter 4

Solution

This chapter presents the developed system call monitoring solution and its various capabilities. The following chapters will cover its usage from the perspective of the three main areas covered by this thesis: detecting a debugger, identifying anomalies and privacy issues.

The developed application is called *"Ptracer"* and can be used to:

- Easily interact with the Linux process tracing interface `ptrace` to capture system calls invocations.
- Learn and enforce a Non-Deterministic Finite State Automata that correlates all the observed system calls sequences and stack traces.
- Provide a deep understanding of the observed system calls to have a higher-level view of the application behaviour.

The project has been developed in C++ and targets the x86_64 and ARMv8-A (also called AArch64) architectures running Linux or Android, different executables have been made for all four possible combinations, and various adaptations are applied to each version. This implies that the project is compatible with most modern Android physical devices, emulators, and any other Linux-based operating systems running on a supported CPU architecture.
Further technical details and instructions on how to build the project are specified in Appendix A.

4.1 Architecture

Since it is necessary to stop the tracee each time a system call is invoked, then using *Ptracer* may imply a performance degradation for the applications, especially in Android, where many system calls are needed to deal with the interactive nature of the system and the need to communicate among multiple components (as described in Section 3.4). Every time a system call notification is received, the tracee will be in a stopped state, and it will be necessary to:

1. Read the CPU registers of the traced process since they are used to specify the identifier of the requested system call and the memory address of its parameters (described in Section 4.2);

2. Extract the sequence of stack frames that lead to the invocation of the system call, if enabled, this may result in multiple read operations from the tracee's memory which come at the cost of a further slowdown (described in Section 4.3);
3. Learning or Enforcing the NFA model depending on what mode has been chosen, and eventually authorizing or not the system call (described in Section 4.4).
4. Analyze the parameters of specific system calls to get an insight into the behaviour of the traced application (described in Section 4.5);

Depending on the desired outcome, it is possible to choose not to extract some data to speed up the traced process; for example, stack traces may not be needed when the goal is only to have an overview of the potential privacy issues.
Despite that, it is necessary to have an internal architecture that allows tracing multiple processes simultaneously and efficiently since even the most straightforward Android application is composed of many of them to optimize its performance.
For these reasons, the tracer has been optimized to keep the traced process in a stopped state for the least amount of time possible and to minimize the number of processes blocked on a system call.

Some of the overhead is unavoidable and directly linked with the notification mechanism performed by the Linux kernel since just pausing a process so often, delivering the signal and performing a context-switch to the tracer is an expensive operation; hence, this part cannot be optimized further without radically changing the monitoring approach with all its consequences (e.g., as mentioned in the future developments Section 6.1).

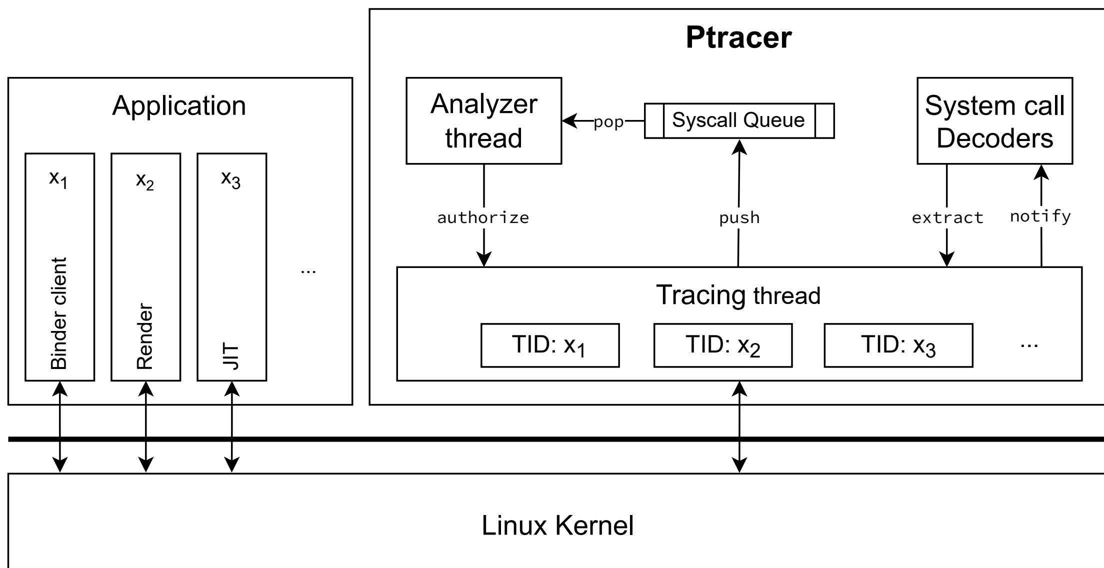


Figure 4.1: Internal architecture of Ptracer in an Android environment

In Figure 4.1, it is possible to see the internal architecture of *Ptracer* where the two main threads are running and tracing a typical Android application. In this case, the Android application is composed of multiple processes (which is a very usual scenario), and since a normal application is depicted, they are all children

of a specialized fork of the Zygote process.
The Tracing thread is responsible for handling all the notifications received from the traced processes via `ptrace`. While the Analyzer component retrieves an internal representation of the received system calls, handles the NFA-based model and has been delegated to make the final decision on whether a system call is authorized.

A concurrent queue of system calls has been created to link the Tracing thread and Analyzer, effectively decoupling them. This is necessary since the two main components will run in two separate threads, and it is desired to keep the Tracing thread as busy as possible handling system call notifications.
This architectural choice has been made to optimize handling applications composed of multiple threads and processes that heavily interact with the kernel, which is the case for Android applications. This came at the cost of slowing down applications that operate on a single thread, which would benefit unifying the two *Ptracer*'s internal processing threads.
There are three different types of notifications produced by the Tracing Thread, each identifying a different type of event:

- **System Call Entry**: this notification type represents the invocation of a new system call by a traced process, it contains the CPU registers and can also embed the stack trace. It will be necessary to have all the information to approve each of them explicitly.
- **System Call Exit**: represents the completion of a system call and carries its return value. These notifications do not require explicit approval since, when received, the action performed by the call has already happened.
- **Process Termination**: received every time a tracee terminates its execution for any reason.

All the notifications mentioned above are specializations of the same common interface, which contains general information like a timestamp and the identifiers of the tracee that has generated it (e.g., its Process ID and Thread ID).
When the Analyzer wants to allow a system call, a signal will be sent to the Tracing thread, whose handler will check the queue of allowed system calls and issue a continue command to the desired Thread Identifier (TID) via `ptrace`. This is necessary since only the specific thread attached to the tracee via `ptrace` can issue a continue command, and the Tracing thread will always be busy waiting for the next system call.
The most immediate solution to this problem would be to create a different tracing thread per process, but this approach has some drawbacks and technical issues. The first problem would be the workload since tracing an application with a high level of parallelism would imply running twice as many threads and processes to monitor it. Secondly, to ensure that no system calls are missed, when a tracee decides to generate a child process, `ptrace` automatically attaches to it. Therefore it would not be possible to detach from it safely to allow a second tracer to take over, at least not without risking missing some system calls during the transition.

Thanks to the separation between a system call's capture and analysis process, it is possible to increment parallelism, enabling fetching the following system call from a different process while another one is being analysed.

To properly analyse the system calls' parameters, it is necessary to read portions of the traced application memory containing the system calls' parameters. This operation needs to happen from the Tracing thread since it is the only one allowed to issue `ptrace` commands. A similar problem has been previously solved by decoupling the two components and using signals.
In this case, a similar approach is not applicable since it would require interrupting the Tracing thread too often to read all the various sections that are part of some system calls.
Therefore, it has been decided to divide the decoders from the tracing thread only from a logical perspective and have them running in the same thread.
The Observer pattern design has been implemented to link these two components. Hence, every decoder will subscribe itself only to the system calls identifiers whose parameters it can parse, and the Tracing thread will be able to notify only the interested decoders.
Since the decoder's implementations run in the Tracing thread, they will be able to extract multiple portions of the tracees' memory and store their analysis results which will be printed and investigated once the application has terminated.
Thanks to this architecture, it is possible to easily create a new decoder just by implementing the `SyscallDecoder` interface and declaring what system calls it shall subscribe to. In this way, it is not necessary to know any detail about the tracing logic to extend *Ptracer*'s analysis capabilities.
More details on this component can be found in Section 4.5.

4.2 System calls monitoring

Leveraging on the Linux process trace interface `ptrace`, it is possible to attach to a specific Identifier for a Process (PID) or Thread (TID) and be notified every time a system call is invoked, or signal is received, hence every time the kernel takes over control. One of the main advantages of this technique is that it cannot be easily evaded and allows modification of system calls parameters and return values.
In Figure 4.2, it is possible to see the point of view that can be obtained using this tracing technique on an Android application; in fact, it allows to be notified every time an action from the kernel is requested either directly by the application code or indirectly by one of the underlying layers. Moreover, *Ptracer* can also extract stack traces, allowing tracking of the sequence of nested functions called from the application code through the ART framework and standard libraries.

Thanks to `ptrace`, it is possible to wait for notifications from a traced process using the library function `waitpid`, which is typically used to wait for the termination of a child process, and in this case, is "abused" to wait for events. It will be necessary to filter those notifications to ensure they are coming from the tracing interface and handled differently if they correspond to the reception of a signal by the tracee, the invocation of a system call, its conclusion or the termination of the entire process.
Moreover, it is also necessary to correctly recognize and implement special cases for some system calls that create new processes or replace the running program.
For example, when a system call like `clone`, `fork` or `vfork` is executed, then the tracee will spawn a new process or thread (depending on the `clone` parameters),

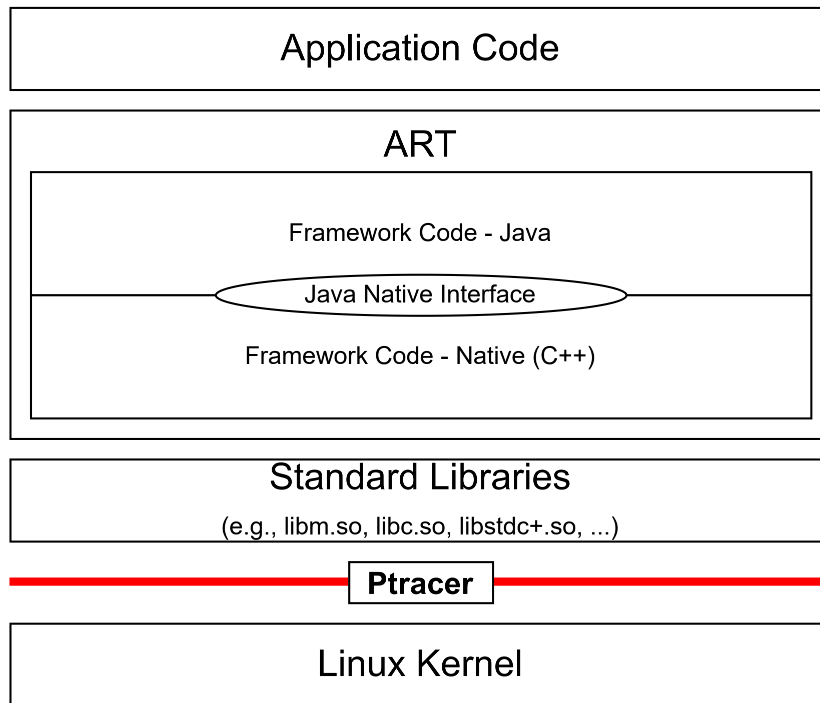


Figure 4.2: Positioning of Ptracer in an Android environment

which will need to be traced too. To do that in the safest way possible, it is possible to leverage on `ptrace`, which, thanks to some initialization options, allows to attach to the generated children of all types automatically.
When an `execve` system call is executed, the running program will be replaced by a new one. This implies that if it is composed of multiple threads, they will all terminate, and only the thread group leader will be left, such behaviour needs to be recognized, and all the data structures used to keep track of the system calls must be modified accordingly.
Moreover, there are special system calls that never return and will generate only one notification at their invocation (contrary to all other calls that generate another one when they terminate). One example is `rt_sigreturn` which should never be called directly and is used to return from a signal handler, hence used every time a signal is received.

To improve security, *Ptracer* also allows placing the traced processes in a "jail" so that if for any reason the tracer dies, then also the tracee will forcefully terminate. In this way, it is possible to prevent the tracee from killing the tracer in an attempt to free itself from its monitoring and control.

Each captured notification from the traced processes will be printed on the standard output, and its internal representation will be made available for further internal processing by the Analyzer component.
In Figure 4.3, it is possible to see an extract output generated from tracing the execution of the command `ls` in an Android environment and disabling the stack trace acquisition.
The execution of this command is single-threaded and does not generate any child process, hence the output is going to be a sequence of pairs of system call entries and their relative exit, if no signals are received. In fact, for each system call invoked by the tracee, the tracer will receive two notifications (three in case of some

special calls), one when the kernel has just received the invocation and the second when it has terminated its execution. In the example, it is possible to see the entry notification of a `write` system call (which identifier in the AArch64 Linux table of system calls is 64), with the Process Identifier (PID) and Thread identifier (SPID) of the process requesting the call.
Moreover, starting from line 8, the parameters passed to the system call are extracted from the CPU registers and reported, together with the Program Counter (PC) and Stack Pointer (SP) in lines 19 and 20.
The subsequent notification type, starting from line 24, is a system call exit, hence the notification received when the kernel has performed the operation and allows to see the return value of the call.

```
1  ----------------- SYSCALL ENTRY START -----------------
2  Notification origin: ls
3  PID: 18470
4  SPID: 18470
5  Timestamp: 1673561040240989
6  NOT Authorized
7  Syscall = write (64)
8  Parameters = {
9     0x0000000000000001
10    0x00000078be7e4989
11    0x0000000000000001
12    0xffffffffffffffff
13    0xffffffffffffffff
14    0x0208001182080800
15    0x0000000000000020
16    000000000000000000
17 }
18 Registers = {
19    PC: 0x0000007a2e962258
20    SP: 0x0000007ffc286370
21    RET: 0x0000000000000001
22 }
23 ------------------ SYSCALL ENTRY STOP ------------------
24 ------------------ SYSCALL EXIT START ------------------
25 Notification origin: ls
26 PID: 18470
27 SPID: 18470
28 Timestamp: 1673561040241834
29 Authorized
30 Return value: 0x00000000000001
31 ------------------ SYSCALL EXIT STOP -------------------
```

Figure 4.3: Partial output of tracing `ls` in Android without capturing the stack trace

4.3 Stack trace extraction

For each system call entry, it is possible to read the stack trace that leads to its generation leveraging on the library `libunwind` [10] on generic Linux-based systems and `libunwindstack` [22] on Android. These libraries offer the possibility to iterate over all the stack frames on a stopped process via `ptrace` and generate a backtrace, effectively fetching the function name and offset from its entry point for each frame.

There are multiple ways to generate a backtrace, which are all dependent on the architecture. One is leveraging on the calling convention that imposes a prologue and epilogue for every function, where the first saves the base stack pointer on the stack and the second restores it when the function execution is over. An example can be seen below:

```
push  rbp
mov   rbp, rsp
;; Function body
pop   rbp
ret
```

The information provided by this prologue and epilogue would allow linking all the stack frames as a list, starting from the most nested one and unwinding the stack upwards. Moreover, thanks to the fact that not only the stack pointer is on the stack but also the return address (used by the final `ret`), it will also be possible to identify the entry point of every function and match it with the relative symbol representing its function name in the dedicated ELF section.

Unfortunately, this first method will not work in all those cases where the frame pointer has been excluded for the sake of optimization. A more modern approach to stack unwinding leverages particular sections of the ELF file format (e.g., the section `.eh_frame` or `.debug_frame`) containing tables with the unwinding information, which can be used to generate a full backtrace.
There are various special cases in unwinding the stack, which can be very platform-specific, for example when a signal is received, a special signal frame is placed on the stack and the process resumed in the signal handler, which will return to a trampoline that will clean the stack and restore the previous situation. In such cases, the unwinding library will need to recognize the trampoline and the special frame to handle it correctly to not confuse frames generated by the handler with the ones generated by the normal program execution.

Android applications run on the Android RunTime (ART) environment, and the large majority are based on languages that can generate Java bytecode with some parts of native code.
Therefore in Android environments, it will be very common to see transitions between native and Java stack frames, hence it is essential to understand both. Moreover, there is an added complexity given by other stack frame formats like Chrome C++ frames, JITed Java frames and system library C++ frames, which can have different call frame information (CFI) formats. Such formats include debug data formats like DWARF and MiniDebugInfo, which describe additional ELF sections containing unwinding data and much more, but also formats like EHABI (Exception Handling ABI for the ARM Architecture), which can help an unwinder in its job.
Thanks to the inclusion of `libunwindstack` and its dependencies from the Android core, it is possible to offload this complexity to the library, which needs to know how to parse the different CFI and move up the stack.

In Figure 4.4 it is possible to see a partial output of an `ioctl` system call invoked by an Android application. It has been necessary to redact some unimportant parts for the sake of readability since the total number of captured stack frames is 135.
The traced application is a simple audio recorder developed for this thesis, it is composed of four buttons: record, stop recording, play the last recording, and stop playing. The reported system call is one of the many generated after the user has pressed the start recording button.
The reported backtrace can be seen from line 9 to line 28, and it is always unwinded starting from the most inner call: in this case, the C wrapper for the `ioctl` system

call, which can be seen at lines 9 and 10.

```
1  ------------------ SYSCALL ENTRY START ------------------
2  Notification origin: attached-process-14711
3  PID: 14711
4  SPID: 14711
5  Timestamp: 1673561963897680
6  NOT Authorized
7  Syscall = ioctl (29)
8  Stack unwinding = {
9   PC 0x000079d54b34d8 Relative PC 0x0000000009d4d8 SP 0000007fcfedfa50 - __ioctl @ 8
10  PC 0x000079d546f2f4 Relative PC 0x000000000592f4 SP 0000007fcfedfa50 - ioctl @ 152
11  PC 0x000079ca76e7ac Relative PC 0x000000000457ac SP 0000007fcfedfb50 - android::IPCThreadState::
        talkWithDriver(bool) @ 288
12  PC 0x000079ca76e958 Relative PC 0x00000000045958 SP 0000007fcfedfbf0 - android::IPCThreadState::
        flushIfNeeded() @ 84
13  PC 0x000079ca77a964 Relative PC 0x00000000051964 SP 0000007fcfedfc10 - android::Parcel::freeDataNoInit() @
        64
14  PC 0x000079ca7861f8 Relative PC 0x0000000005d1f8 SP 0000007fcfedfc70 - android::Parcel::freeData() @ 16
15  ...
16  PC 0x000079dc6fe870 Relative PC 0x00000000000870 SP 0000007fcfee1740 - it.matteodegiorgi.audiorecorder.
        MainActivity.CheckPermissions @ 32
17  ...
18  PC 0x000079dc6fed3c Relative PC 0x00000000000d3c SP 0000007fcfee1af0 - it.matteodegiorgi.audiorecorder.
        MainActivity.startRecording @ 0
19  ...
20  PC 0x000079dc6fe8dc Relative PC 0x000000000008dc SP 0000007fcfee1e90 - it.matteodegiorgi.audiorecorder.
        MainActivity.lambda$onCreate$0$it-matteodegiorgi-audiorecorder-MainActivity @ 0
21  ...
22  PC 0x000079dc6fe790 Relative PC 0x00000000000790 SP 0000007fcfee2230 - it.matteodegiorgi.audiorecorder.
        MainActivity$$ExternalSyntheticLambda0.onClick @ 4
23  ...
24  PC 0x0000007286a62c Relative PC 0x0000000082f62c SP 0000007fcfee47a0 - com.android.internal.os.ZygoteInit.
        main @ 2188
25  ...
26  PC 0x000079d9c1bfb4 Relative PC 0x000000000c0fb4 SP 0000007fcfee4ca0 - android::AndroidRuntime::start(...)
        @ 836
27  PC 0x000057c299858c Relative PC 0x0000000000258c SP 0000007fcfee4d90 - main @ 1336
28  PC 0x000079d545e7dc Relative PC 0x000000000487dc SP 0000007fcfee5f00 - __libc_init @ 96
29 }
30 Parameters = {
31  0x000000000000003c
32  0x00000000c0306201
33  0x0000007fcfedfb68
34  0x00000076c8bee008
35  0000000000000000
36  0x00000077336274e8
37  0000000000000000
38  0000000000000000
39 }
40 Registers = {
41  PC: 0x00000079d54b34d8
42  SP: 0x0000007fcfedfa50
43  RET: 0x000000000000003c
44 }
45 ------------------ SYSCALL ENTRY STOP ------------------
```

Figure 4.4: Partial output from *Ptracer* of a system call coming from an Android AudioRecorder including its stack trace

Moving up the stack, it is possible to see that the Android IPC interface has requested the system call as part of the transmission of a Parcel (previously explained in Section 3.4), this can be observed from line 11 to 14.
As it is possible to see from the different function name formats, from lines 16 to 22, a portion of stack frames is generated from Java code. More precisely and moving in an upwards direction on the stack, the function `CheckPermissions` is making sure that the correct Android privileges have been granted (line 16), which has been called by the `startRecording` method (line 18), which is the reg-

istered `onClick` listener for its related button, as it is possible to see at line 22.
Near the top of the stack (from lines 24 to 26), it is possible to see that the application is a specialized fork of the Zygote process, which is typical for Android applications.
Eventually, at lines 27 and 28, it is possible the usual entry point for every process, hence the `main` function called from the `libc` initialization wrapper.
For each stack frame, it is possible to see also:

- **The Program Counter (PC)**: also known as the Instruction Pointer (IP), is the address of the instruction after invoking the function generating the underlying stack frame, hence the return address of the called procedure.
- **The Relative PC**: is the displacement between the entry point of the function generating the stack frame and the previously described PC.
- **Stack Pointer (SP)**: is the address to the top of the stack frame, it can also be intended as the stack base pointer.

In Figure 4.5 it is possible to see a diagram showing where the three pointers mentioned above refer to. In this case, a simple example program called "EvenOdd" has been used, it reads a number from standard input and prints on standard output if it is even or odd.
The analysed system call is a `read` performed by `scanf`, and from the diagram, we can see the stack trace that has been extracted. Looking at the highlighted line in the stack trace, it is possible to see PC pointing to the instruction after the call to `scanf`, since it is where the function `__isoc99_scanf` will return when completed. Relative PC is the difference between PC and the address of the first instructions of the function, in this case, it can be computed as follows:

$$\text{0x5555555551B6 (PC)} - \text{0x555555555169 (main entry point)} = \text{0x4D} = 77$$

SP refers to the top of the stack allocated by the `main` function call, it is the address that is stored on the stack at the beginning of a function call (the `RBP` register), when used, or retrieved via other ELF sections.

Ptracer always requests the name of the function that has generated a stack frame, and in the proposed examples, the stack unwinder has always managed to retrieve one. Whether the executable or the used libraries do not contain enough information to retrieve such information, the relative PC will be used since both PC and SP can change between various executions due to the Address Space Layout Randomization (ASLR, better explained in Section 3.3).

As seen from the previous example, it is possible to use the stack trace to get an insight into the context where a system call takes place and its purpose. This information will provide a critical advantage in detecting anomalies as described in Section 5.2

Figure 4.5: Diagram showing where the various information extracted by *Ptracer* point in the example program EvenOdd

4.4 NFA Model

When the system call and stack traces acquisition has been completed, it is possible to produce a representation of the observed behaviour in the form of a Non-Deterministic Finite state Automata (NFA). More specifically, the Non-Deterministic version of the automaton has been chosen because of the nature of the treated transitions. Such choice allows not to represent incorrect combinations, which otherwise would need to all lead to an error sink state, and hence have a simplified model given the large number of observed states. Moreover, system calls that generate children (e.g., a `fork`) will result in two ε-transitions since, from that moment on, there will be two machines performing computational activities.
For these reasons, it has been necessary to use an NFA (instead of its deterministic counterpart) and chosen not to opt for more complex models, e.g., based on push-down automaton, since their expressivity power is not needed for the current scope. This implies that the model will be able to describe a regular grammar (type 3).

In the generated NFA, every state expresses an observed system call together with the stack trace that leads to it, and each transition represents the observed succession of system calls learnt.
It will be necessary to undergo a learning phase where the NFA will be built and refined for each subsequent execution. In this phase, it is essential to test all the possible execution paths of the application until the model converges, hence there is no new transition or state.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5    int n;
6
7    printf("Insert a number: ");
8    scanf("%d", &n);
9
10   if (n % 2 == 0) {
11     printf("Even number\n");
12   }
13   else {
14     printf("Odd number\n");
15   }
16
17   return 0;
18 }
```

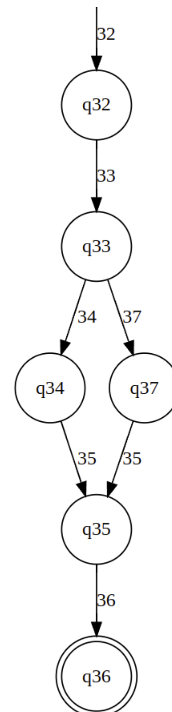


Figure 4.6: Source code of the program EvenOdd with the resulting NFA

In Figure 4.6, it is possible to see the source code of the previously discussed program called *EvenOdd* and an extract of its corresponding NFA on an x86_64 Linux system. Only the most important states are shown, hence the part covering the program initialization (e.g., the execution of `_start` and `__libc_start_main`)

ID	System Call	Stack Trace
33	0 (read)	((read, 17), (__IO_file_underflow, 390), (__IO_default_uflow, 54), (___vfscanf, 3256), (___isoc99_scanf, 178), (_main, 77), (___libc_init_first, 144), (___libc_start_main, 138), (__start, 37))
34	1 (write)	((__write, 20), (__IO_file_write, 45), (__IO_file_setbuf, 256), (__IO_do_write, 25), (__IO_file_overflow, 259), (_puts, 346), (_main, 102), (___libc_init_first, 144), (___libc_start_main, 138), (__start, 37))
35	8 (lseek)	((lseek64, 11), (__IO_file_sync, 126), (__IO_default_xsgetn, 245), (__IO_file_setbuf, 14), (__IO_adjust_column, 1039, (_erand48_r, 546), (_exit, 32), (___libc_init_first, 151), (___libc_start_main, 138), (__start, 37))
36	231 (exit_group)	((_exit, 49), (_erand48_r, 562), (_exit, 32), (___libc_init_first, 151), (___libc_start_main, 138), (__start, 37))
37	1 (write)	((__write, 20), (__IO_file_write, 45), (__IO_file_setbuf, 256), (__IO_do_write, 25), (__IO_file_overflow, 259), (_puts, 346), (_main, 119), (___libc_init_first, 144), (___libc_start_main, 138), (__start, 37))

Figure 4.7: Redacted example of system calls and stack traces mapped by the *Mapper* function for EvenOdd

have been omitted for the sake of clarity.
In such a simple case, the automata only needed two learning iterations before reaching convergence since one covered odd numbers and the other even numbers.

The `libc` library functions `printf` and `scanf` boil down to, respectively, a `write` and a `read` system call on the Standard Output and Standard Input file descriptors. They can be seen in Figure 4.7 where the state `q33` is performed at line 8 of the source code and states `q34` and `q37` corresponds to the two `printf` at lines 11 and 14.
Moreover, it is possible to see from `q34` and `q37` stack traces how the compiler has optimised these two calls converting them into two `puts`, which is more efficient and limited than `printf`.
Thanks to the inclusion of stack traces, two distinct possible transitions are starting from the state `q33`, both triggered by a `write` system call but with different stack traces given the different position of `printf` in `main`, as it can be observed from the stack traces.
Eventually, the state `q36` is given by the system call `exit_group`, and it is marked as final since it causes the process termination.

Every state in the NFA corresponds to a 3-tuple with the following content:

$$(\mathit{ExecutableName}, \mathit{SystemCall}, \mathit{StackTrace})$$

It is necessary to specify also *ExecutableName*, whose value is a string since that can change during program execution (e.g., due to an `execve` system call), and it will be useful to prevent errors when selecting the NFA to use for multiple executions.
SystemCall is represented by its numeric identifier, uniquely defined per CPU architecture, hence: $\mathit{SystemCall} \in \mathbb{N}$. An example of this identifier can be seen in Figure 4.4 at line 7 between parenthesis or in Figure 4.7 in the System Call column.
The *StackTrace* element is a tuple of variable length and composed itself by 2-tuples of the type $(\mathit{FunctionName}, \mathit{Offset})$. In the previously mentioned example reporting the stack backtrace output, *FunctionName* and *Offset* have been separated by an `@` symbol in the stack unwinding section.
The stack trace tuple can be expressed as follows:

$$((\mathit{FunctionName}_0, \mathit{Offset}_0),\ (\mathit{FunctionName}_1, \mathit{Offset}_1),\ ...)$$

Each state has assigned an integer identifier to ease automata management and handling through a mapping function. Hence, allowing the separation of two different concerns, on one side, the creation and manipulation of the automata, and on the other, understanding stack traces and comparing them.
To do that a *Mapper* function has been defined as follows:

$$\mathit{Mapper}:\ \{(\mathit{ExecutableName}, \mathit{SystemCall}, \mathit{StackTrace})\} \longmapsto \mathbb{N}_{>0}$$

The identifier 0 is not part of the function codomain since it always represents the initial state and does not correspond to any system call or stack trace.
This function has been implemented using a Hash Map data structure to retrieve the state's identifier, given its tuple, efficiently. Such operation will need to be executed as fast as possible to reduce the performance deterioration of the tracee

since it will take place every time a decision is made whether an observed state is authorized or not.
This function is injective, therefore, two 3-tuple that are equivalent will map to the same identifier.

The generated NFA can be formally represented using the following 5-tuple:

$$\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

Where each element consists of the following:

- $Q = \{\texttt{q0},\ \texttt{q1},\ ...\}$ It is a finite set and contains all the possible states of the automata. Thanks to the *Mapper* function, every received system call entry notification will be mapped to a positive integer, and its value will be collected in this set.

- $\Sigma = \{1,\ 2,\ 3,\ ...\}$ Represents the finite set of input symbols. In this case, it corresponds with all the identifiers generated by the *Mapper* function during the learning phase, hence with all the unique acquired states.

- $\delta : Q \times \Sigma \mapsto \mathcal{P}(S)$ This function defines the transitions from one state to another, given the current state and input symbol. Where $\mathcal{P}(S)$ denotes the power set of Q since, theoretically, there can be multiple destination states for each pair of state and input symbol. In our case, this happens only when the process flow forks, action represented by two ε-transitions from the same state. These transitions are learnt from the succession of states observed during the learning phase, making the anomaly detection process as good as its learning phase.

- $q_0 = \texttt{q0}$ The initial state does not correspond with any system call or stack trace and can be interpreted as the program launch or the initial attach operation on a running process.

- $F \subset Q$ The set of final states is going to contain all those states where the tracee has terminated its execution during the learning phase. These states are represented by special termination notifications from the tracee and can happen either because of a system call in the `exit` family or a signal. When the model is enforced, a tracee termination that does not happen in any of these states will raise an anomaly alert.

Now that the used model has been formally described, it is necessary to define one algorithm to learn it by constructing the automata and another to enforce it by checking whether a new system call notification is allowed. The first will be used during the learning phase, which is required before using the anomaly detector, and the second will need to traverse the automata and keep track of the current state for each tracee.

The automata construction process can be done by first collecting the following sets of data:

- The set of identifiers of the observed system calls entry notifications equal to Σ and $Q - \{0\}$. Created by iterating over all states retrieved through the relative queue (shown in Figure 4.1), computing their identifier via the *Mapper* function, and then storing them.

- The set of final states F, composed of all the states where a tracee is placed when its termination notification is received.
- The allowed transitions expressed by the δ function, which can be described as a list of 3-tuples, each of the form: $(Source, Symbol, Destination)$. Every transition not contained in this list is not authorised and will need to be handled by the Authorizer component. The list can be created by keeping track of what is the last state for each tracee and, for each new state, compose the tuple with the identifier of the old state as *Source* and the identifier returned by the *Mapper* function for the new state as both *Symbol* and *Destination*. In the case of a child-generating system call, it will be necessary to create two ε-transitions, which symbolize the bifurcation of the execution flow.

The above sets are constructed and augmented over multiple learning iterations until convergence is reached. At this point, it is possible to construct the NFA model since all five elements of its tuple can be defined given the above information.

The second necessary algorithm is necessary to determine whether a state is authorised. This algorithm not only needs to produce a *yes* or *no* answer but also needs to keep track of the position of each tracee in the NFA and identify the type of anomaly among the following:

- **Unknown State Anomaly**: happens every time a state that has never been seen during the learning phase occurs. This means that the stack trace and system call combination has never been observed before; e.g., this can happen if a backdoor is triggered.
- **Unknown Transition Anomaly**: occurs when the received state has been previously mapped by the *Mapper* function, but a transition from the current state to it has never been observed. E.g., this can happen in the case of a Return-Oriented Programming (ROP) attack when an ROP gadget is called.
- **Anomalous Termination**: occurs every time a tracee terminates its execution without being in a terminal state. This anomaly can happen when a termination signal is sent, or a critical error happens; hence the program does not follow its typical termination path.

In Algorithm 1, it is possible to see the pseudo-code of this algorithm. The Hash Map *currentStates* is used to keep track of the traced processes' position using as key the Thread Identifier (TID), which is unique overall to the whole Operating System.
To simplify handling the automata, it has been decided not to include ε-transitions in its representation but only implement them in this algorithm. This can be done by performing the equivalent operation of adding a new machine computing the NFA starting from the state that has generated the new child, as can be seen in line 14.

Leveraging the previously defined algorithms to learn and enforce the model, it is possible to use *Ptracer* to detect anomalies successfully.
When one is detected, the Authorizer component will raise an alert if it is a Termination Anomaly or prompt the user for a choice in case it is an Unknown State or Transition anomaly.

Algorithm 1 Implementation of the enforce algorithm

```
1: procedure IsAuthorised(state)
2:     if Authorizer.learningMode then
3:         ▷ It will be used for the offline model generation
4:         stateStorage.add(state)
5:         return AUTHORISED
6:     end if
7:     ▷ Obtain the current tracee's state
8:     current ← currentStates[state.TID]
9:     if current = ∅ then
10:        if stateStorage.isEmpty() then
11:            currentStates[state.TID] ← Model.initialStates
12:        else if state is the first state of a child then
13:            ▷ Set the child current position to the same as its parent
14:            parent ← The state in childGenerators that generates state.TID
15:            currentStates[state.TID] ← Mapper.find(parent)
16:        else                          ▷ The state comes from an unknown tracee
17:            return ERROR
18:        end if
19:    end if
20:    if state.isTerminationNotification() then
21:        ▷ Check if the tracee is in a final state
22:        if current ∩ Model.FinalStates ≠ ∅ then
23:            return AUTHORISED
24:        else
25:            return TERMINATION_ANOMALY
26:        end if
27:    end if
28:    symbol ← Mapper.find(state)
29:    if symbol = NOT_FOUND then
30:        return UNKNOWN_STATE_ANOMALY
31:    end if
32:    future ← Model.transition(current, symbol)
33:    if future = ∅ then
34:        return UNKNOWN_TRANSITION_ANOMALY
35:    end if
36:    currentStates[state.TID] ← future
37:    if state.SystemCall ∈ {clone, fork, vfork} then
38:        ▷ The child's TID is not yet known, store the current position to place
   the child in the future
39:        childGenerators.add(state)
40:    end if
41:    return AUTHORISED
42: end procedure
```

The user will be able to choose one of the following actions:

- Terminate the traced program: all the tracees are forcefully terminated together with their tracers;

- Add a new transition: applicable only in case of an Unknown Transition anomaly, the model will be modified by adding the missing transition;
- Add a new state and transition: applicable in case of an Unknown State anomaly, a new state will be inserted in the model together with a transition that leads to it from the current state.

The produced model can be very useful for detecting anomalies in a program and getting a visual representation of the program's behaviour.
It will be leveraged in Section 5.2 for detecting anomalies in Android applications and will provide a data structure that can be expanded for debugging detection, which will be discussed further in Section 5.3.

4.5 System call decoders

In all those occasions where it is desired to get an insight into a program's behaviour, it is helpful to analyse its interactions with the kernel since each of them needs to be explicitly requested via a system call. This implies that any Linux process needs to perform a system call to request any action that does not involve calculations and memory operations in its already pre-allocated virtual memory.
Currently, Linux offers more than 300 system calls, which differ per CPU architecture and perform various actions; hence, it is important to isolate the most meaningful ones as those that could give the best insight into the performed actions.
For example, if the tracee is requesting the operating system to open a file, then it is of interest to capture its full path and what read/write operations are requested. Similarly, when an Internet connection is requested, having information like the destination address and what data has been transmitted and received can provide valuable clues on the final intentions of the analysed program. Moreover, in an Android application, capturing inter-process communications via the Binder will provide a view into what data are accessed by the application.

System calls with similar effects have been grouped together, and the most insightful groups have been selected:

- **Open a file path**: this group includes all the system calls which accept a file path and, if the requester has enough permissions, return an identifier that can then be used to perform operations on the file or directory. Moreover, this group also contains all those system calls that can read or write from a file descriptor. This family of calls provides insight into the files used by the analysed program.
- **Initiate a connection**: this group includes system calls used to connect various sockets (e.g., TCP, UDP, UNIX, and more). This analysis will give a clear picture of what network communications are requested by the application.
- **Usage of the ptrace interface**: This group covers various usages of the Linux process tracing interface. Its main usage is to take control of a process for various purposes like monitoring and debugging, but it can also be used as an anti-debugger technique (as seen in [1]). Analysing this call will allow knowing if the tracee is interacting with any external processes.

- **Android Binder IPC Framework**: In an Android environment, every communication between activities, system components, and services needs to pass through the Binder IPC interface (more information on it can be found in Section 3.4). Thanks to this design, there is a central point that all the inter-process communications have to go across. Analysing such communications will grant insight into the program's interactions with the Android system, components and services (e.g., the microphone).

- **Execute a new program**: This group includes system calls able to completely replace the currently running program, fundamentally altering the application behaviour. They are mostly used to run different programs and will be useful to know if any external command has been executed.

The composition of these different views, which, taken in isolation, may not mean a lot, will result in a greater understanding of the program's behaviour and provide a holistic view.

Thanks to its privileged position between the kernel and the processes, *Ptracer* also offers the possibility of analysing the actions performed by these system calls, which can give clues on the analysed program's final intentions. Such capability is particularly useful in detecting privacy issues, which will be discussed further in Section 5.4.
Moreover, the developed tool also allows for retrieving parameters passed to the system call, which are specified in standard CPU registers according to the system call convention, which is architecture-specific. Such a feature will be leveraged to provide a deeper insight into the requested action.

This kind of analysis has been embedded in the architecture as the component "System Call Decoders" which contains various decoders, each able to analyze a specific family of system calls. Each implements a common interface that can be seen in Figure 4.8, where every declared method is purely virtual and must be implemented.
The two `decode` methods will be used by the Tracing thread to notify a specific decoder that a call of its interest has been received or has just been completed. Being able to receive both notification types is necessary to be able to see the passed parameters in the entry notification, which are extracted from the CPU registers and may be altered during the call execution, and see the return value in the exit notification.
When the traced program's execution is terminated, every decoder will be asked to print on standard output a report of the observed behaviour via the `printReport` function.

```
1 class SyscallDecoder {
2 public:
3   virtual bool decode(const ProcessSyscallEntry& syscall) = 0;
4   virtual bool decode(const ProcessSyscallExit& syscall) = 0;
5   virtual void printReport() const = 0;
6 };
```

Figure 4.8: Interface implemented by every system call decoder

It is desired that all the logic concerning a particular system call family is contained only within its corresponding decoder; hence, each of them shall be able to

subscribe to a list of system call identifiers it can analyse. Moreover, it is essential to minimize as much as possible the time spent in decoders since, in the meantime, it is not possible to process new notifications because decoders are executed by the Tracing thread (this architectural choice has been discussed in Section 4.1).
These requirements have been satisfied by introducing a mapper from System Call to Decoder. This component will be the external entry point to all the decoders and handles all the subscription requests. It is based on a Hash Map data structure that maps a system call identifier to the list of decoders that previously subscribed to it; in addition, it allows fetching this list in constant time. It will receive every System call entry and exit notification from the Tracing thread, and it will forward it to any interested decoder.

Together with all the standard information extracted for every system call (e.g., CPU registers, stack backtrace, timestamp), every entry and exit notification also contains a handler to the specific Tracer instance that has generated it. It provides methods to copy portions of the tracee memory into the tracer, operations used by all the decoders that wish to follow a memory pointer specified in the system call parameters (e.g., to a structure specifying other parameters).

In Figure 4.9, it is possible to see a diagram of the described decoders' architecture. The diagram shows how the decoders interact with the tracing threads and how they get notified of a new system call from the tracing thread.

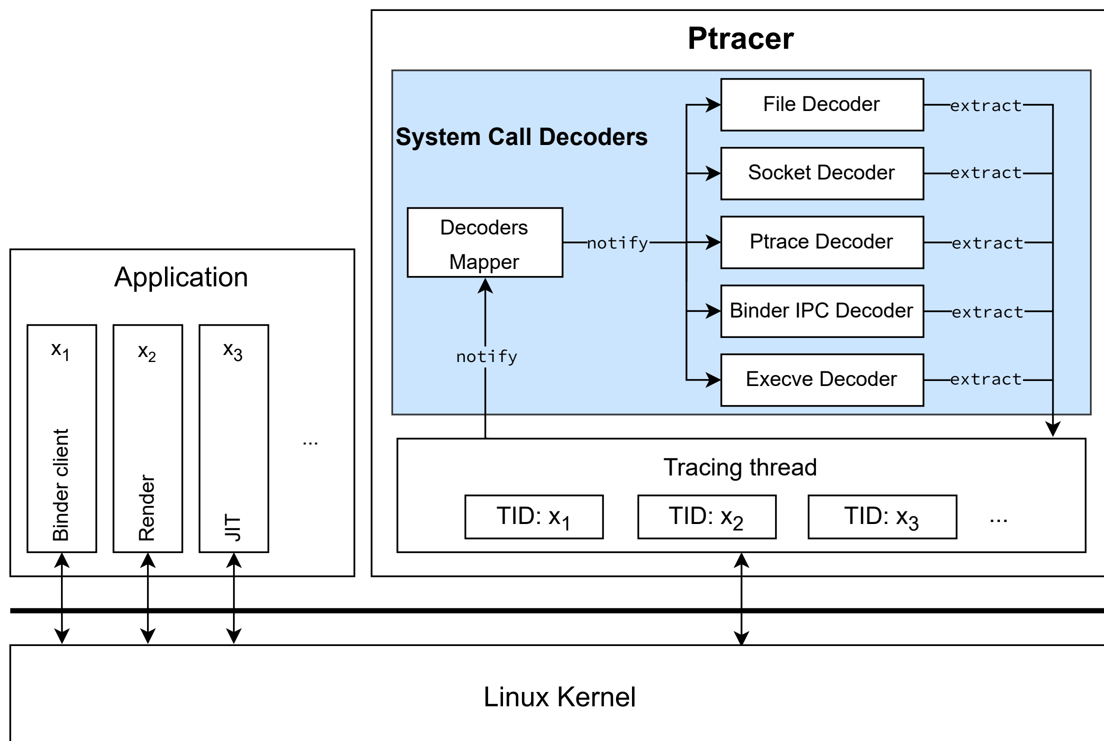


Figure 4.9: Architecture of the System Call Decoders

At any moment, it is possible to request a full report for each decoder through the Decoders Mapper. By default, *Ptracer* will request the final report after the termination of all the traced processes.
The report will be subdivided per Process ID since most of the analysed system calls have a meaning only in that context, e.g., the returned file descriptors are unique per PID. The only exception is Ptrace Decoder which will be subdivided

per Thread ID since the Linux tracing interface always refers to that level of granularity.

The implementation of Decoders in *Ptracer* still has a lot of potential developments in terms of widening the set of supported system calls and deepening the extracted insights. Hence, the current implementation does not aim to be complete but to provide a solid foundation for future developments, which will be discussed in Section 6.2.

4.5.1 File Decoder

The File Decoder handles system calls that can open and perform I/O operations on file descriptors. The current implementation covers the following system calls categories for the x86_64 and ARMv8 architecture:

- Open a file and bind it to a file descriptor: `open` (available only in x86_64), `openat`, `openat2`, `name_to_handle_at`, and `creat`.
- Read the content from a file descriptor: `read`, `recvfrom`, and `pread64`.
- Write data to a file descriptor: `write`, `sendto`, and `pwrite64`.

In the following paragraphs, these three categories will be better analysed.

To bind a file descriptor to a data source, it is necessary to use one of the system calls in the first category. Even though their behaviour is slightly different and, for some, can be heavily influenced by the specified flags, they all follow a common line of action: given a path to a file, if the requester has enough permissions, they will return an identifier that can then be used to perform operations on it.
From their function signature specified below, it is possible to see that each accepts a string parameter named `pathname`.

```
int open(const char *pathname, int flags, mode_t mode);

int openat(int dirfd, const char *pathname, int flags, mode_t mode);

// Standard libraries do not provide a wrapper for this system call
long syscall(SYS_openat2,
            int dirfd,
            const char *pathname,
            struct open_how *how,
            size_t size);

int name_to_handle_at(int dirfd,
                      const char *pathname,
                      struct file_handle *handle,
                      int *mount_id,
                      int flags);

int open_by_handle_at(int mount_fd,
                      struct file_handle *handle,
                      int flags);

int creat(const char *pathname, mode_t mode);
```

For each of the system calls mentioned above, this decoder registers itself to the entry notification to extract the string pointed by `pathname` and the exit notification to retrieve the returned file descriptor or the error code. It will then be possible

to match the file descriptor with other I/O operations that the application might request.

The second category of system calls analysed by this decoder can read data from a file descriptor. Hence, they all require specifying a file descriptor and a pointer to the buffer that can accommodate the read data.
Their specifications can be seen below:

```
ssize_t read(int fd, void *buf, size_t count);

ssize_t recvfrom(int sockfd,
                 void *buf,
                 size_t len,
                 int flags,
                 struct sockaddr *src_addr,
                 socklen_t *addrlen);

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

The File Decoder will register itself to the entry, and exit notification of the system calls mentioned above.
When an entry notification is received, it will be necessary to store the file descriptor and buffer address since the buffer will be empty until the system call is successfully executed, and this is the only moment when these parameters can be retrieved.
Once the exit notification is received, it will be possible to reuse the previously stored parameters to extract the buffer and use the return value of the system call, which represents the number of bytes read as its length.
Data read by these calls will be stored on files in the file system, they will be placed in a folder having as name the Process ID (PID), since file descriptors are unique per PID, and concatenated with the timestamp since file descriptors can be reused once freed.

The last category of system calls whose analysis is delegated to this decoder can write data to a file descriptor. All these calls accept a file descriptor and the address of a buffer containing the data to write, together with its size in bytes.
Their specifications can be seen below:

```
ssize_t write(int fd, const void *buf, size_t count);

ssize_t sendto(int sockfd,
               const void *buf,
               size_t len,
               int flags,
               const struct sockaddr *dest_addr,
               socklen_t addrlen);

ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

The decoder will register itself to the entry notifications of the system calls mentioned above. Once a notification is received, it can directly extract the buffer that the application wants to write to the file descriptor together with its length. The extracted data will be stored on the file system in a similar fashion as the read data.

In Figure 4.10, it is possible to see the final report generated by the decoder after monitoring the execution of the following bash command:

```
$ cat /etc/passwd
```

Since this command has a very simple and reproducible behaviour in every Linux distribution, to generate the attached report it has been run on a generic one.
From this execution, we can see the program initialization from lines 6 to 9, which happens before giving control to the `main` function. In this case, the archive that contains the cached locations of shared libraries has been opened in line 6, it is reasonable to assume that it has provided the location of `libc`, which is the standard C library, since it has been opened and read just after that to link it dynamically.
In lines 10 and 11, the file path `/etc/passwd` has been opened, assigned the file descriptor 3, and its whole content read.
Eventually, we can see from line 4 that the same number of bytes as those read from `/etc/passwd` has been written to standard output.
Some file descriptors have been opened and not read using the usual system calls traced by this decoder, for example, the file at line 6 has been mapped in the virtual address space using a `mmap` system call, this operation does not allow to know what portions of data has been accessed and will be analysed in future developments.

```
1  ------------------ FILE DECODER START ------------------
2  File Descriptor: 0 <---> STDIN
3  File Descriptor: 1 <---> STDOUT
4  Write content extracted in: "./FileDecoder/455143/1-write-1675561338553", bytes: 2308
5  File Descriptor: 2 <---> STDERR
6  File Descriptor: 3 <---> /etc/ld.so.cache
7  File Descriptor: 3 <---> /usr/lib/libc.so.6
8  Read content extracted in: "./FileDecoder/455143/3-read-1675561338466", bytes: 2400
9  File Descriptor: 3 <---> /usr/lib/locale/locale-archive
10 File Descriptor: 3 <---> /etc/passwd
11 Read content extracted in: "./FileDecoder/455143/3-read-1675561338551", bytes: 2308
12 ------------------ FILE DECODER STOP ------------------
```

Figure 4.10: Final report generated by the Open Decoder

This group provides insight into what paths are accessed by the analysed program or, in case an error occurs, which ones it is trying to access. This will be useful to know what files are accessed by an application to be able to identify potential privacy intrusions.
Future implementations will expand this set and improve the analysis capabilities.

4.5.2 Socket Decoder

Initiating a new connection to an external source or, more generally, starting to transmit data via the network requires creating and operating on a socket. Hence, it is possible to know what external hosts the application is communicating with by analysing all those system calls that perform actions on them.
This section will describe all the system calls handled by the Socket Decoder and will focus only on TCP and UDP sockets using IPv4 or IPv6 addresses, and Unix sockets. The rationale behind this choice is limiting the decoder complexity given the large amount of supported address families and protocols by the Linux kernel and the fact the selected ones are used for communicating over the Internet and with system components.
Protocols like Bluetooth are handled by their relative Hardware Abstraction Layer, hence their sockets are not directly accessible by applications but are indirectly

accessible via the Binder IPC, an interaction which will be analysed in Section 4.5.4.

To create a socket, it is necessary to use the `socket` system call, whose C wrapper can be seen below:

```
int socket(int domain, int type, int protocol);
```

It accepts as a parameter the communication domain (e.g., IPv4, Unix, etc.), the socket type which defines the communication semantics (e.g., two-way connection-based stream of bytes, datagrams, etc.), and eventually, the protocol to use (e.g., TCP, UDP, etc.).
Its return value is a socket file descriptor, which will be used by the subsequent calls operating on the created socket. This decoder will not analyse this system call since it does not provide useful information and needs to be used in conjunction with the following calls to be meaningful.

Once the socket is created, it is possible to "assign a name to it", which consists of binding it to an address. This is done in all those cases where it is desired to receive data at an address using the system call `bind`, this call has the following syntax:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

It accepts as parameters the socket file descriptor generated by the previous call, a structure containing the address information and its length.
The `sockaddr` parameter is a generic container for more specialised structures that can describe different address formats (e.g., IPv4, IPv6, Unix, etc.). This structure declaration can be seen below:

```
struct sockaddr {
  sa_family_t     sa_family;     // Address family
  char            sa_data[14];   // Address data e.g. sockaddr_in
};
```

To know which is used among the various implementations of address types, it is necessary to check the value of the member `sa_family`.
If its value is equal to the constant `AF_INET`, then that instance of `sockaddr` can be interpreted as a `sockaddr_in`, which allows specifying an IPv4 address (32 bits) together with the destination port (16 bits). This data structure can be seen below:

```
struct sockaddr_in {
  sa_family_t     sin_family;    // e.g. AF_INET
  unsigned short  sin_port;      // e.g. htons(3490)
  struct in_addr  sin_addr;      // 32 bit unsigned integer
  unsigned char   sin_zero[8];   // Pad to size of sockaddr
};
```

Instead, if the value of `sa_family` is `AF_INET6`, then it is possible to reinterpret the structure as a `sockaddr_in6` which allows specifying an IPv6 address and port together with other protocol-specific parameters. Below it is possible to see this data structure:

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;   // AF_INET6
  unsigned short  sin6_port;     // port number
  uint32_t        sin6_flowinfo; // IPv6 flow information
  struct in6_addr sin6_addr;     // IPv6 address
  uint32_t        sin6_scope_id; // Scope ID (new in 2.4)
```

```
};
struct in6_addr {
  unsigned char   s6_addr[16];   // IPv6 address
};
```

Eventually, if its value is `AF_LOCAL`, then a `sockaddr_un` structure can be expected, which contains the path to the UNIX socket on the file system. Its specification can be seen below:

```
struct sockaddr_un {
  sa_family_t     sun_family;    // AF_UNIX
  char            sun_path[108]; // Pathname
};
```

Tracing the `bind` system call will allow knowing the local address and port where the application is expecting to receive data.
This analysis is necessary since, in sockets that are not connection-based, it is enough to bind an address to start waiting for data on it, therefore, this call is the only chance to extract the local listening address.

Once an address has been bound to the socket, if its type is connection-based (e.g., TCP or Unix), it is possible to set it in listening mode (via the `listen` system call, not analysed here) and begin to accept connections using the `accept` system call. This call accepts as parameters the socket file descriptor, a structure that will be populated with the peer socket information (e.g., the IP of the remote host) and an optional flags parameter to specify various options.
The Linux kernel offers the following two system calls, their only difference is the possibility to specify the flags parameter, as can be seen below:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t len);

int accept4(int sockfd, struct sockaddr *addr, socklen_t len, int flags);
```

By analysing these system calls, it will be possible to know what hosts connect to the traced application by extracting the address specification.

In case it is desired to act as a client and connect to a remote host (e.g., establish a TCP connection to a remote server), or more generically, the underlying socket is connection-based, it will be necessary to connect the socket actively using the `connect` system call.
If the socket is connection-less, then this call can be used to set the default destination and allow incoming data only from a specific address.
This call accepts as parameters a socket file descriptor and the usual `sockaddr` structure containing the destination address.
Below it is possible to see the specification of the system call wrapper exposed by the standard C library and the data structure it uses.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Analysing this call will allow having a clear picture of what external parties the application is requesting to communicate with.

The Socket Decoder registers itself to the entry, and exit notification of `bind`, `accept`, `accept4`, and `connect` system calls. When such entry notifications are received, the decoder will extract the address definition `sockaddr`, which is specified as the second parameter in the analysed calls. This address will be used to generate a final report which specifies, for each socket, all the addresses involved.

In Figure 4.11, it is possible to see the final report generated by the Socket Decoder after tracing the following bash command, which fetches Google's homepage, on a generic Linux system:

```
$ curl http://www.google.com
```

```
1  ------------------ SOCKET DECODER START ------------------
2  Local socket, to address: [/var/run/nscd/socket], error: -2, No such file or directory
3  Local socket, to address: [/run/systemd/resolve/io.systemd.Resolve], error: -2, No such file or directory
4  IPv4, to address: [8.8.8.8]:53
5  IPv4, to address: [142.250.184.100]:80
6  IPv6, to address: [2a00:1450:4002:405::2004]:80, error: -101, Network is unreachable
7  IPv4, to address: [142.250.184.100]:80, error: -115, Operation now in progress
8  ------------------ SOCKET DECODER STOP ------------------
```

Figure 4.11: Final report generated by the Socket Decoder on a generic Linux system

In this example, `curl` needs first to resolve the provided domain name using a resolver (`www.google.com`), connect to the resulting IP address, send an HTTP request and then print the response on standard output.
Resolving the domain name is done via the C standard library, which first tries to reach the Name Service Cache Daemon (NSCD) Unix socket in line 2, then the SystemD Resolver in line 3, and falls back to the primary Google DNS server in line 4.
The requested domain name offers an IPv4 and IPv6 address, and they are both tried, the latest can be seen in line 6 but will fail because the used machine does not have any IPv6 connectivity, eventually, the request to the other IP address succeeds in line 5.

```
1  ------------------ SOCKET DECODER START ------------------
2  Socket File Descriptor: 7 <---> Local socket, to address: [/dev/socket/dnsproxyd]
3  Socket File Descriptor: 6 <---> Local socket, to address: [/dev/socket/fwmarkd]
4  Socket File Descriptor: 5 <---> IPv4 Internet protocol, to address: [142.251.36.36]:80, error: -115,
       Operation now in progress
5  Socket File Descriptor: 6 <---> Local socket, to address: [/dev/socket/fwmarkd]
6  ------------------ SOCKET DECODER STOP -------------------
```

Figure 4.12: Final report generated by the Socket Decoder on an Android system

In Figure 4.12 it is possible to see another example where the same command has been run on a phone using Android 12.
In this case, the used resolver differs, as can be seen in line 2, since every Android application delegates this task to the system resolver available via the UNIX socket `/dev/socket/dnsproxyd`.
Moreover, in Android, it is necessary an extra step that can be seen in lines 3 and 5. The FwmarkServer is contacted via its UNIX socket at `/dev/socket/fwmarkd`, its role is enforcing the permission system, hence only allowing connections from applications that have properly requested permission to do so (e.g., via the Android Manifest) and handling packet routes, e.g., allowing an application to bypass a VPN. This is done by setting a Firewall Mark (fwmark) on the socket if the application is allowed and using its value to determine the packets' routes.
In this case, the application communicates the socket file descriptor to allow it to be "marked" by the service and performs a second call after the connection for reporting purposes.
Eventually, in line 4, the connection is requested and made.

This decoder will be useful to gain a picture of what external parties over the network the analysed application is communicating with. Such information is going to be very important to identify potential information exfiltration.

4.5.3 Ptrace Decoder

This decoder aims to analyse all the various usages of the Linux process tracing interface by monitoring the `ptrace` system call. Its main usage is to take control of a process for various purposes like monitoring and debugging, but it can also be used as an anti-debugger technique (as seen in [1]).
Analysing this call will allow knowing if some anti-debugging techniques are being used or if the tracee is interacting with any external processes since this behaviour could represent an attempt to evade analysis.
Moreover, tampering with the return value of this system call could allow deceiving the tracee into believing that no tracer is present.

All the requests to the process tracing interface are made through the `ptrace` system call, whose syntax can be seen below:

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

This decoder will need to register itself to the entry notifications of this system call to be able to store the most important passed parameters: the request type and the target Thread ID (TID). These two pieces of information will be displayed in the final report, and it will be possible to see in a human-readable format what operations the application is performing via `ptrace` and on what process.

```
1  ------------------ PTRACE DECODER START ------------------
2     ...
3  Command: PTRACE_SEIZE on SPID: 494775
4  Command: PTRACE_INTERRUPT on SPID: 494775
5  Command: PTRACE_LISTEN on SPID: 494775
6  Command: PTRACE_SYSCALL on SPID: 494775
7  Command: PTRACE_GETSIGINFO on SPID: 494775
8  Command: PTRACE_SYSCALL on SPID: 494775
9  Command: PTRACE_GET_SYSCALL_INFO on SPID: 494775
10 Command: PTRACE_SYSCALL on SPID: 494775
11 Command: PTRACE_GETEVENTMSG on SPID: 494775
12 Command: PTRACE_SYSCALL on SPID: 494775
13 Command: PTRACE_GET_SYSCALL_INFO on SPID: 494775
14 Command: PTRACE_SYSCALL on SPID: 494775
15 Command: PTRACE_GET_SYSCALL_INFO on SPID: 494775
16    ...
17 Command: PTRACE_SYSCALL on SPID: 494775
18 Command: PTRACE_GET_SYSCALL_INFO on SPID: 494775
19 Command: PTRACE_SYSCALL on SPID: 494775
20 Command: PTRACE_GET_SYSCALL_INFO on SPID: 494775
21 Command: PTRACE_SYSCALL on SPID: 494775
22 Command: PTRACE_SYSCALL on SPID: 494775
23 ------------------ PTRACE DECODER START ------------------
```

Figure 4.13: Final report generated by the Ptrace Decoder

In Figure 4.13, it is possible to see the final report generated by the Ptrace Decoder after tracing the following bash command:

```
$ strace cat /etc/passwd
```

The specified command has been run on a phone using Android 12, and it launches `strace`, a generic tool to display the system calls requested by a process, in this case, it will spawn a child process, which will run the `cat` command and trace it

while it prints the content of `/etc/passwd` on standard output.
To be able to let `strace` correctly attach to its child, it has been necessary to use the *Ptracer* options that disable automatically attaching to every child of traced processes since only one tracer per process is admitted.
The shown report has been redacted for the sake of clarity, only the part concerning the traced command has been left and truncated in the middle, given the large number of system calls.
From the report, it is possible to see the initial `PTRACE_SEIZE` command in line 3, which is used to attach to the target process (the `cat` command in this case).
Subsequently, `strace` will put its tracee in a stopped state (line 4) and start listening for events (line 5), e.g., the invocation of a system calls or a signal.
After this initial setup, a loop is started, where for each notification received from the tracee, if it is a signal, then the request type `PTRACE_GETSIGINFO` is used to gather information about it, if it is a system call, then the necessary data is fetched using the request `PTRACE_GET_SYSCALL_INFO`. Then, after retrieving the necessary information, the request `PTRACE_SYSCALL` is used to allow the tracee to proceed until the next event.

This decoder will be useful to detect usages of the Linux process tracing interface, have an insight into the dynamics present among the traced processes, and detect reverse engineering prevention techniques.

4.5.4 Binder IPC Decoder

Whether an Android application needs to acquire data from a sensor or trigger an action in another component (e.g., open the dialer), it will need to perform inter-process communications (IPC). In any Android environment, every communication between activities, system components, and services needs to pass through the Binder IPC interface (more information can be found in Section 3.4) since it handles passing messages between applications, synchronization, shared memory, shared object's lifespans, and remote procedure calls.
Thanks to this design, there is a central point that all such communications need to cross, often considered the "heart" of Android. Therefore, intercepting and analysing data crossing such a core part of the system is bound to provide valuable insight into the application behaviour.
Please, note that the data structures reported in this section refer to Android 12 and might differ in other versions.

The Binder exposes the character device `/dev/binder`, and every process can perform read and write operations on it using the `ioctl` system call. Such communications are performed by the `libbinder.so` library and abstracted away from the user.
From a user perspective, there are multiple services that the application can interact with, and they all have their Java interface that describes what the available method calls are.

To be able to capture and analyse all the communications with the Binder, it is necessary to intercept the `ioctl` system call, which has the following signature:

```
int ioctl(int fd, unsigned long request, ...);
```

Its generic usage is manipulating the underlying device-specific parameters of special files, allowing controlling many operating characteristics in a way that cannot

be expressed by regular system calls.
It requires specifying a file descriptor for the target device, a device-dependent request code, and, optionally, a third parameter (traditionally called `char *argp`) as an untyped pointer to an additional data structure.
The passed parameters are highly dependent on the target device since each device accepts operations of different types and natures, e.g. setting the transfer rate on a serial port.

In the case of the Android Binder, the file descriptor will always be associated with the device `/dev/binder`, while the request type and additional parameter are variable and dependent on each other.
There are various request types, most of which are used during the initial handshake with the Binder or to detach from it. It has been decided not to analyse requests that are always performed and do not carry any information for our purposes.
Therefore only the most important request type is analysed: `BINDER_WRITE_READ`, which is used to write and read commands to and from the Binder.
When this request type is used the third parameter needs to be a pointer to a structure of type `binder_write_read`, its specification can be seen below:

```
struct binder_write_read {
  binder_size_t write_size;
  binder_size_t write_consumed;
  binder_uintptr_t write_buffer;
  binder_size_t read_size;
  binder_size_t read_consumed;
  binder_uintptr_t read_buffer;
};
```

The main role of this structure is to point to the read and write buffers as well as carry their size and number of bytes already consumed by either the Binder or the client.
Once this structure is fetched from the tracee's memory, it will be necessary to extract the read and write buffers. The write buffer is used to send data to the Binder driver, and the read buffer is used to receive data.
Therefore, during the system call entry notification, it will be necessary to extract `write_size` bytes starting from the address `write_buffer + write_consumed`.
Since the read buffer will be written by the Binder driver, it is necessary to extract it when the `ioctl` system call terminates. Therefore, its address will be saved during the entry notification to be then extracted when the call will return since there is no guarantee that system call parameters are preserved after its execution. The buffer that will be extracted during the exit notification is pointed by `read_buffer + read_consumed`, and its size is `read_size` bytes.

The buffers thus obtained contain a list of concatenated commands sent from the Binder to the application, each has a different meaning and might be followed by additional data structures.
Each of these commands is part of the `binder_driver_commaand_protocol` enumeration, which has been reported below:

```
enum binder_driver_command_protocol {
  BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
  BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
  BC_ACQUIRE_RESULT = _IOW('c', 2, __s32),
  BC_FREE_BUFFER = _IOW('c', 3, binder_uintptr_t),
```

```
  BC_INCREFS = _IOW('c', 4, __u32),
  BC_ACQUIRE = _IOW('c', 5, __u32),
  BC_RELEASE = _IOW('c', 6, __u32),
  BC_DECREFS = _IOW('c', 7, __u32),
  BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
  BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
  BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
  BC_REGISTER_LOOPER = _IO('c', 11),
  BC_ENTER_LOOPER = _IO('c', 12),
  BC_EXIT_LOOPER = _IO('c', 13),
  BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_handle_cookie),
  BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_handle_cookie),
  BC_DEAD_BINDER_DONE = _IOW('c', 16, binder_uintptr_t),
  BC_TRANSACTION_SG = _IOW('c', 17, struct binder_transaction_data_sg),
  BC_REPLY_SG = _IOW('c', 18, struct binder_transaction_data_sg),
};
```

In the reported enumeration, it is possible to see also the expected data structures for each command type, for example, `BC_INCREFS` is followed by an unsigned 32-bit integer (`__u32`), and `BC_INCREFS_DONE` is followed by a structure of type `binder_ptr_cookie`.
The decoder will need to iterate over the buffer, check what command types have been specified, and parse the following bytes according to the data structure associated with each command. Moreover, it also needs to take into consideration that there can be multiple commands in a single buffer.
Although the implemented decoder can parse all the commands mentioned above, those that contain transactions are the most informative: `BC_TRANSACTION`, `BC_REPLY`, `BC_TRANSACTION_SG`, and `BC_REPLY_SG`. These commands are always followed by a transaction definition, in the case of the first two, its type is the structure `binder_transaction_data`, and in the others, it will be `binder_transaction_data_sg`. The specification of these structures can be seen below:

```
struct binder_transaction_data {
  union {
    __u32 handle;
    binder_uintptr_t ptr;
  } target;                   // 32-bit handle or pointer
  binder_uintptr_t cookie;    // Used to detect mismatched handles
  __u32 code;                 // Transaction code, built-in or application-defined
  __u32 flags;                // Defined in enum transaction_flags
  pid_t sender_pid;           // Process identifier of sender
  uid_t sender_euid;          // UID of message sender
  binder_size_t data_size;    // Buffer size pointed by data
  binder_size_t offsets_size; // Size of offsets pointer by data
  union {
    struct {
      binder_uintptr_t buffer;  // Transaction data
      binder_uintptr_t offsets; // Offsets to flat_binder_object's
    } ptr;
    __u8 buf[8];
  } data;
};

struct binder_transaction_data_sg {
  struct binder_transaction_data transaction_data;
  binder_size_t buffers_size;   // Total size of all buffers
};
```

The second structure is newer and contains the first with the addition of the parameter `buffers_size`, in fact, they have the same end goal: deliver a transaction to the Binder. Their only difference lies in the transfer of the transaction object from the client through the Binder to its destination since the second one enables using Scatter/Gather I/O.
This optimization speeds up the Binder operations significantly and has been discussed in Section 3.4.

Once extracted from the tracee's memory, it is necessary to parse the two obtained structures `binder_transaction_data`, an operation that will be done in the entry or exit notification, as previously discussed.
There are two pointers in every transaction structure, the first is `data.buffer`, and the second is `data.offsets`. Their value is valid only if their size in bytes, respectively, `data_size` for the first and `offsets_size` for the second, is greater than 0. If valid, the decoder will also extract and store those memory areas to represent them while printing the final report.
These two memory areas will be used to transfer data and object references about the desired inter-process method call or response, their combination represents a Parcel: a message container commonly used in Android environments.

The data format used inside `buffer` is highly variable, and depending on the end party the application is interacting with, there are hundreds of available services that are exposed to the Binder, and each can contain custom methods and data classes.
For this reason, it has been decided to limit the provided insight into the exchanged Parcels, future developments will be proposed in Section 6.2 to provide a deeper understanding of methods and data classes used by each available Android system service.

The buffer memory can contain information regarding what method has been invoked and the passed parameters, as well as external object references. In fact, it also has the role of object reference management, which is handled via the pointer `data.offsets`, which points to a memory area containing `offsets_size` / 8 elements, where each of them can be interpreted as an offset from `data.buffer` for a separate Binder Object.
Such structures reference live objects of various natures, e.g., file descriptors or handles to remote objects, and all their possible types can be seen below:

```
enum {
  BINDER_TYPE_BINDER = B_PACK_CHARS('s', 'b', '*', B_TYPE_LARGE),
  BINDER_TYPE_WEAK_BINDER = B_PACK_CHARS('w', 'b', '*', B_TYPE_LARGE),
  BINDER_TYPE_HANDLE = B_PACK_CHARS('s', 'h', '*', B_TYPE_LARGE),
  BINDER_TYPE_WEAK_HANDLE = B_PACK_CHARS('w', 'h', '*', B_TYPE_LARGE),
  BINDER_TYPE_FD = B_PACK_CHARS('f', 'd', '*', B_TYPE_LARGE),
  BINDER_TYPE_FDA = B_PACK_CHARS('f', 'd', 'a', B_TYPE_LARGE),
  BINDER_TYPE_PTR = B_PACK_CHARS('p', 't', '*', B_TYPE_LARGE),
};
```

For each offset, the decoder computes the position in the buffer where the Binder Object is located, for example, the first object can be referenced as follows: `transaction.data.buffer + *(transaction.data.offsets[0])` and then cast the obtained pointer to one of the data structures corresponding with the specific object type.
To do that, their generic interface is used to acquire their type, which can be seen below:

```c
struct binder_object_header {
  __u32 type;
};
```

Hence, the first 4 bytes are matched with the previously mentioned types to be able to identify the referenced structure. The associations between object types and data structures can be seen below, together with the specifications of all the structures:

```c
// Used for BINDER_TYPE_BINDER, BINDER_TYPE_WEAK_BINDER, BINDER_TYPE_HANDLE and
    BINDER_TYPE_WEAK_HANDLE
struct flat_binder_object {
  struct binder_object_header hdr;
  __u32 flags;
  union {
    binder_uintptr_t binder;
    __u32 handle;
  };
  binder_uintptr_t cookie;
};
// Used for BINDER_TYPE_FD
struct binder_fd_object {
  struct binder_object_header hdr;
  __u32 pad_flags;
  union {
    binder_uintptr_t pad_binder;
    __u32 fd;
  };
  binder_uintptr_t cookie;
};
// Used for BINDER_TYPE_PTR
struct binder_buffer_object {
  struct binder_object_header hdr;
  __u32 flags;
  binder_uintptr_t buffer;
  binder_size_t length;
  binder_size_t parent;
  binder_size_t parent_offset;
};
// Used for BINDER_TYPE_FDA
struct binder_fd_array_object {
  struct binder_object_header hdr;
  __u32 pad;
  binder_size_t num_fds;
  binder_size_t parent;
  binder_size_t parent_offset;
};
```

Even though it would have been possible to go deeper into some of these data structures (since they can reference other memory areas), it has been decided not to do that since it would imply an additional slowdown of the traced application without providing any further insight.

In Figure 4.14, it is possible to see a diagram of the extracted memory areas in a typical scenario where the client sent a transaction and the driver is responding, acknowledging its competition and with a full transaction reply.
The diagram shows the relationship between the various memory areas, as previously discussed, and how they cascade from the initial `binder_write_read` structure all the way to the Parcel buffer and offsets.

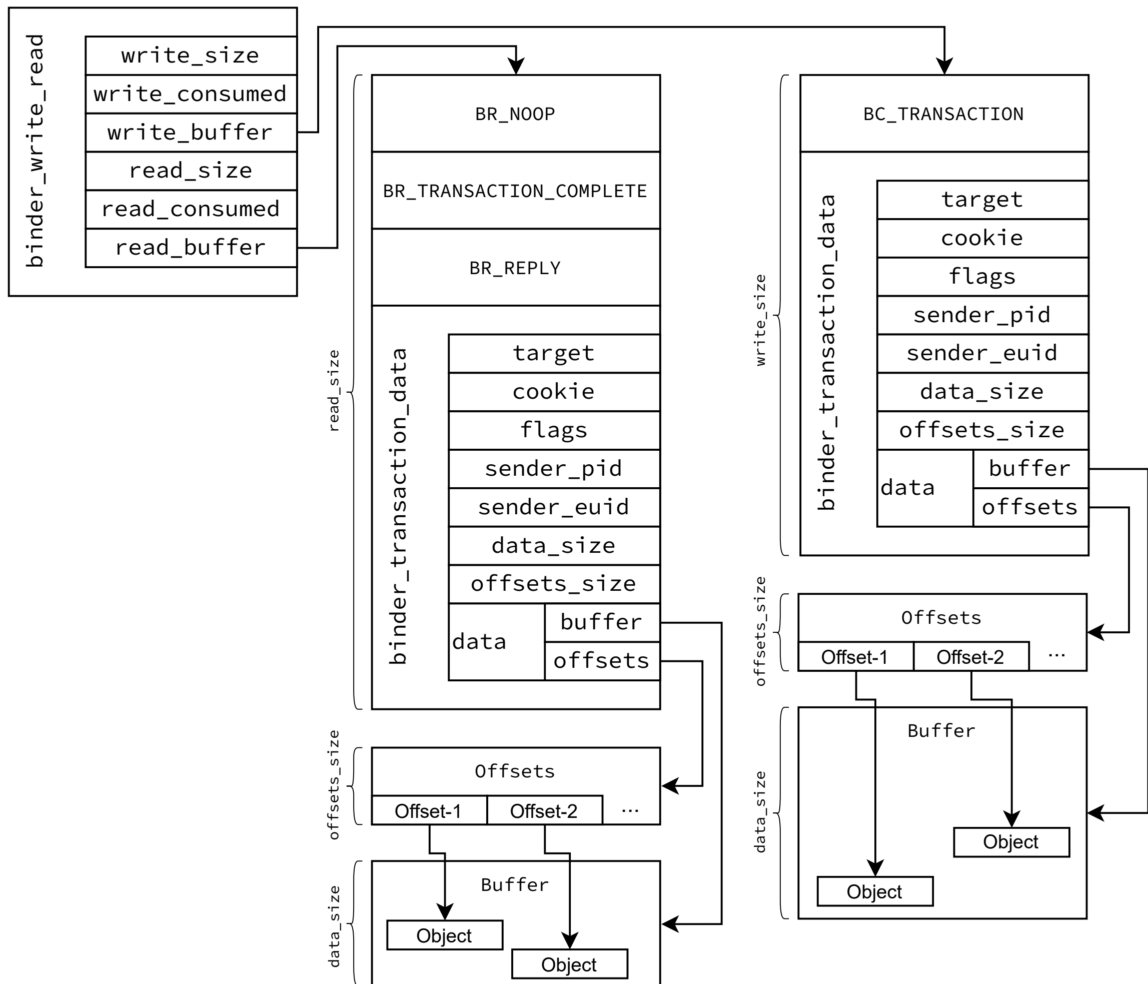


Figure 4.14: Representation of the Read/Write buffer structure extracted by the Android Binder Decoder

To demonstrate how the Binder Decoder works, it has been decided to trace a simple interaction with the Android Binder where the phone service is invoked, and a number is dialled. This can be done using the `service` Android command, which is used to invoke remote procedures in one of the exposed services via the Binder.
To do that, it is necessary to use the `phone` service, which exposes a method to dial a number passed as a parameter.
The full list of services registered in the Binder can be visualized using the following command, where the service in question has been highlighted:

```
$ service list
```

When run using the tested device (running Android 12), the above command produces a list of 229 services associated with their interface declaration.
A snippet of this output can be seen in Figure 4.15 where in line 8 there is the one that will be used.

Now it is possible to visualize all the methods exposed by the `phone` service opening its AIDL interface definition from the Android 12 source code repository.
AIDL stands for “Android Interface Definition Language”, it allows the definition of a service’s programming interface and lets Android handle all the necessary operations to invoke methods and send back the return values remotely.

```
1    ...
2  146   performance_hint: [android.os.IHintManager]
3  147   permission: [android.os.IPermissionController]
4  148   permission_checker: [android.permission.IPermissionChecker]
5  149   permissionmgr: [android.permission.IPermissionManager]
6  150   persistent_data_block: [android.service.persistentdata.IPersistentDataBlockService]
7  151   phone:  [com.android.internal.telephony.ITelephony]
8  152   pinner: []
9  153   platform_compat: [com.android.internal.compat.IPlatformCompat]
10 154   platform_compat_native: [com.android.internal.compat.IPlatformCompatNative]
11 155   pocket: [android.pocket.IPocketService]
12 156   power: [android.os.IPowerManager]
13   ...
```

Figure 4.15: Snippet of the services exposed via the Binder with their interface declaration

A snippet from this interface containing the interested method has been reported in Figure 4.16 where it can be seen in line 13.

```
1  package com.android.internal.telephony;
2
3  /* ... */
4
5  interface ITelephony {
6    /**
7     * Dial a number. This doesn't place the call. It displays the Dialer screen.
8     * @param number the number to be dialed. If null, this
9     * would display the Dialer screen with no number pre-filled.
10    */
11   @UnsupportedAppUsage
12   void dial(String number);
13
14   /* ... */
15 }
```

Figure 4.16: Snippet of the `ITelephony` interface definition containing the method that will be called

Now that the service interface is known, it is possible to compose the command to invoke the `dial` method remotely, passing as a parameter a string containing the number that should be typed into the dialer. The used command will be the following:

```
$ service call phone 1 s16 "000"
```

Where the argument `call` is used to specify that it is desired to perform a remote method invocation, subsequently, the target service name is specified: `phone`, followed by the method number, which in this case is 1 since it is the first in its interface, and eventually the specification of the parameter that should be passed in the form of a UTF-16 string (the standard Java character set) specified via the `s16` specifier.

Thanks to the Binder Decoder, it is possible to have an insight into all the interactions that the above command has triggered. To better explain them, they will be split into sections extracted from the final report generated by the decoder.
In fact, the client will first need to discover the target service, get a handle to communicate with it, and eventually perform the requested method call and fetch

the result.

```
------------------ BINDER CALL START ------------------
Sent:
 Protocol: 0x40406300 (BC_TRANSACTION)
  Target: 0x0 (0)
  Code: 2
  Flags: 16 (TF_ACCEPT_FDS)
  Buffer pointer: 0x7b09b3ec30 (528443763760), Data size: 88
  Buffer content:
  0x7b09b3ec30: 00 00 00 80 ff ff ff ff 54 53 59 53 1a 00 00 00 ........ TSYS....
  0x7b09b3ec40: 61 00 6e 00 64 00 72 00 6f 00 69 00 64 00 2e 00 a.n.d.r. o.i.d...
  0x7b09b3ec50: 6f 00 73 00 2e 00 49 00 53 00 65 00 72 00 76 00 o.s...I. S.e.r.v.
  0x7b09b3ec60: 69 00 63 00 65 00 4d 00 61 00 6e 00 61 00 67 00 i.c.e.M. a.n.a.g.
  0x7b09b3ec70: 65 00 72 00 00 00 00 00 05 00 00 00 70 00 68 00 e.r..... ....p.h.
  0x7b09b3ec80: 6f 00 6e 00 65 00 00 00                         o.n.e...
  Interface: android.os.IServiceManager

Received:
 Protocol: 0x720c (BR_NOOP)
 Protocol: 0x7206 (BR_TRANSACTION_COMPLETE)
 Protocol: 0x80407203 (BR_REPLY)
 Target: 0x0 (0)
 Sender EUID: 1000
 Buffer pointer: 0x7a199dc000 (524415778816), Data size: 32
 Buffer content:
 0x7a199dc000: 00 00 00 00 85 2a 68 73 13 01 00 00 01 00 00 00 .....*hs ........
 0x7a199dc010: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 0c ........ ........
 Offsets pointer: 0x7a199dc020 (524415778848), Offsets size: 8
 Offsets content:
 0x7a199dc020: 04 00 00 00 00 00 00 00                         ........
 Offset 0:
  Type: 0x73682a85 (BINDER_TYPE_HANDLE)
  Flags: 275 (BINDER_BUFFER_FLAG_HAS_PARENT|FLAT_BINDER_FLAG_PRIORITY_MASK|FLAT_BINDER_FLAG_ACCEPTS_FDS)
  Handle: 1
------------------ BINDER CALL STOP -------------------
```

Figure 4.17: Binder Decoder output showing the application request for a handle to the `phone` service

After the usual initial steps to initialize the data exchange with the Binder, the application will request a handle to the `phone` service to then be able to call the desired method, such operations can be seen in Figure 4.17. This operation is done by querying the `android.os.IServiceManager`, as can be seen in line 15.

```
package android.os;

/* ... */

interface IServiceManager {
  /* ... */

  /**
   * Retrieve an existing service called @a name from the service
   * manager. Non-blocking. Returns null if the service does not exist.
   */
  @UnsupportedAppUsage
  @nullable IBinder checkService(@utf8InCpp String name);

  /* ... */
}
```

Figure 4.18: Snippet of the `IServiceManager` interface definition containing the method that will be called

The Service Manager is a special service, it is the official Android endpoint mapper and gets started at the system start-up, if it fails, the whole system crashes. The main role of this service is managing all the other services by maintaining a mapping between a service name and a proxy handle to the service.
The invoked method is specified in the `code` field and can be seen in line 5.
In this case, it corresponds to the second method of the `IServiceManager` AIDL interface definition, which can be seen in Figure 4.18.

From the received response, we can see that the transaction has been completed (in line 19), and the reply starts from line 20.
The Service Manager has replied with an external object specified in the offsets (starting from line 30), which, given its type (`BINDER_TYPE_HANDLE`) carries a handle that can be used to reach the `phone` service. The handle value can be seen in line 33 and corresponds to "1".

```
1  ------------------ BINDER CALL START ------------------
2  Sent:
3    Protocol: 0x40046304 (BC_INCREFS)
4    Handle: 1
5  ------------------ BINDER CALL STOP -------------------
6  ------------------ BINDER CALL START ------------------
7  Sent:
8    Protocol: 0x40046305 (BC_ACQUIRE)
9    Handle: 1
10 ------------------ BINDER CALL STOP -------------------
11 ------------------ BINDER CALL START ------------------
12 Sent:
13   Protocol: 0x40086303 (BC_FREE_BUFFER)
14   Buffer: 0x7a199dc000 (524415778816)
15 ------------------ BINDER CALL STOP -------------------
```

Figure 4.19: Binder Decoder output showing the claim of a handle and a request to free a buffer

At this point, the client is interested in declaring its interest in the received handle and that it should be kept alive, such operation can be seen in Figure 4.19.
This is done by first increasing the count of weak references to the handle and then the count of strong references, since this is the first time that the service is referenced by this application. These operations can be seen in line 3, where the count of weak references is increased via the `BC_INCREFS` command and then in line 8, where the strong references count is increased via the `BC_ACQUIRE` command.
Eventually, the client can send a `BC_FREE_BUFFER` command (in line 13) to inform the driver that it can free the kernel buffer used for the reply.

In Figure 4.20, it is possible to see the client requesting the canonical interface descriptor to the previously acquired handle.
Starting from line 3, it is possible to see the request sent by the application, it does not need any buffer since its intent is clear by the specified flag in line 5 (`INTERFACE_TRANSACTION`).
The response comes directly from the `phone` service and can be seen starting from line 9, it contains its interface descriptor (extracted in line 22), which corresponds with its fully-qualified Java interface name.
Eventually, just like before, the client informs the driver that the buffer at the address `0x7a199dc000` used for the reply can be freed (operation omitted in the report snippet).

```
1  ------------------ BINDER CALL START ------------------
2  Sent:
3   Protocol: 0x40406300 (BC_TRANSACTION)
4    Target: 0x1 (1)
5    Code: 1598968902 (INTERFACE_TRANSACTION)
6    Flags: 16 (TF_ACCEPT_FDS)
7
8  Received:
9   Protocol: 0x720c (BR_NOOP)
10  Protocol: 0x7206 (BR_TRANSACTION_COMPLETE)
11  Protocol: 0x80407203 (BR_REPLY)
12   Target: 0x0 (0)
13   Sender EUID: 1001
14   Buffer pointer: 0x7a199dc000 (524415778816), Data size: 88
15   Buffer content:
16   0x7a199dc000: 29 00 00 00 63 00 6f 00 6d 00 2e 00 61 00 6e 00 )...c.o. m...a.n.
17   0x7a199dc010: 64 00 72 00 6f 00 69 00 64 00 2e 00 69 00 6e 00 d.r.o.i. d...i.n.
18   0x7a199dc020: 74 00 65 00 72 00 6e 00 61 00 6c 00 2e 00 74 00 t.e.r.n. a.l...t.
19   0x7a199dc030: 65 00 6c 00 65 00 70 00 68 00 6f 00 6e 00 79 00 e.l.e.p. h.o.n.y.
20   0x7a199dc040: 2e 00 49 00 54 00 65 00 6c 00 65 00 70 00 68 00 ..I.T.e. l.e.p.h.
21   0x7a199dc050: 6f 00 6e 00 79 00 00 00                         o.n.y...
22   Interface: com.android.internal.telephony.ITelephony
23   Offsets pointer: 0x7a199dc058 (524415778904), Offsets size: 0
24 ------------------ BINDER CALL STOP -------------------
```

Figure 4.20: Binder Decoder output showing the application acquiring the interface descriptor of the `phone` service

Now the client has all the necessary information to perform the desired remote procedure call, which requires: having a handle for the remote service, its interface descriptor, and knowing the interface method number and the parameters to pass to it.

```
1  ------------------ BINDER CALL START ------------------
2  Sent:
3   Protocol: 0x40406300 (BC_TRANSACTION)
4    Target: 0x1 (1)
5    Code: 1
6    Flags: 16 (TF_ACCEPT_FDS)
7    Buffer pointer: 0x7b09b3d0a0 (528443756704), Data size: 112
8    Buffer content:
9    0x7b09b3d0a0: 00 00 00 80 ff ff ff ff 54 53 59 53 29 00 00 00 ........ TSYS)...
10   0x7b09b3d0b0: 63 00 6f 00 6d 00 2e 00 61 00 6e 00 64 00 72 00 c.o.m... a.n.d.r.
11   0x7b09b3d0c0: 6f 00 69 00 64 00 2e 00 69 00 6e 00 74 00 65 00 o.i.d... i.n.t.e.
12   0x7b09b3d0d0: 72 00 6e 00 61 00 6c 00 2e 00 74 00 65 00 6c 00 r.n.a.l. ..t.e.l.
13   0x7b09b3d0e0: 65 00 70 00 68 00 6f 00 6e 00 79 00 2e 00 49 00 e.p.h.o. n.y...I.
14   0x7b09b3d0f0: 54 00 65 00 6c 00 65 00 70 00 68 00 6f 00 6e 00 T.e.l.e. p.h.o.n.
15   0x7b09b3d100: 79 00 00 00 03 00 00 00 30 00 30 00 30 00 00 00 y....... 0.0.0...
16   Interface: com.android.internal.telephony.ITelephony
17
18 Received:
19  Protocol: 0x720c (BR_NOOP)
20  Protocol: 0x7206 (BR_TRANSACTION_COMPLETE)
21  Protocol: 0x80407203 (BR_REPLY)
22   Target: 0x0 (0)
23   Sender EUID: 1001
24   Buffer pointer: 0x7a199dc000 (524415778816), Data size: 4
25   Buffer content:
26   0x7a199dc000: 00 00 00 00                                    ....
27   Offsets pointer: 0x7a199dc008 (524415778824), Offsets size: 0
28 ------------------ BINDER CALL STOP -------------------
```

Figure 4.21: Binder Decoder output showing the remote method call of the `dial` method

In Figure 4.21, it is possible to see the procedure call to the `dial` method via the Binder.

Starting from line 3, there is the transaction sent to the `phone` service via the previously acquired handle (which is specified as the transaction `target.handle` in line 4).
In line 16, it is possible to see that the targeted interface is the one exposed by `ITelephony`, and in line 5, which prints the `code` field, the method number is specified, in this case, it is the number 1 corresponding to `dial`.
Moreover, in the hexadecimal dump of the buffer data, the string parameter passed to the method can be seen immediately after the interface descriptor in line 15.
Since the invoked function has void return type, the reply starting from line 21 does not carry any additional data.

At this point, the desired method has been invoked successfully, and the dialer pops up on the phone screen with the requested number ("000") already typed.

```
1  ----------------- BINDER CALL START -----------------
2  Sent:
3    Protocol: 0x40086303 (BC_FREE_BUFFER)
4      Buffer: 0x7a199dc000 (524415778816)
5  ----------------- BINDER CALL STOP ------------------
6  ----------------- BINDER CALL START -----------------
7  Sent:
8    Protocol: 0x40046306 (BC_RELEASE)
9      Handle: 1
10 ----------------- BINDER CALL STOP ------------------
11 ----------------- BINDER CALL START -----------------
12 Sent:
13   Protocol: 0x40046307 (BC_DECREFS)
14     Handle: 1
15 ----------------- BINDER CALL STOP ------------------
```

Figure 4.22: Binder Decoder output showing the release of a handle and a request to free a buffer

It is now necessary to release the used resources, an operation that can be seen in Figure 4.22.
First, in line 3, the driver will be informed that it can release the buffer used for the reply request.
Then the previously received handle to the `phone` service with value 1 will be released, to do that, first the count of its strong references is decreased using the `BC_RELEASE` command in line 8, and then the count of weak references is decreased using `BC_DECREFS` in line 13.

In Android, every inter-process interaction requires interacting with the IPC Binder, to the point that an application can do very little without it. Monitoring interactions between Binder and applications will prove to be very useful in identifying privacy issues and will be discussed further in Section 5.4.

4.5.5 Execve Decoder

This decoder aims to analyse the usage of the system calls that can replace the currently running program with an executable from the file system. The system calls able to do that are the following: `execve` and `execveat`.

When an application would like to execute an external executable, the typical behaviour is generating a child process (e.g., via a `fork`) and then using one of these system calls to inform the kernel that the running program should be substituted.

This operation has many security implications since it is not only used for legit cases, but it is often a way for an attacker to spawn a remote shell and take control of the machine. Moreover, it also has implications in terms of privacy issues detection since running external executables can give clues on what is the relationship between the various application components and be interpreted as a way to evade some monitoring techniques.

The specifications of the two system calls analysed by this decoder are the following:

```
int execve(const char *pathname, char *const argv[], char *const envp[]);

int execveat(int dirfd,
             const char *pathname,
             const char *const argv[],
             const char *const envp[],
             int flags);
```

They both take as input parameters a string `pathname`, which specifies the location of the executable that shall replace the currently running program, an array of strings `argv`, which communicates the list of arguments that should be passed to the executable, and a second array of strings `envp` with the environment variables for the new process image.
Their difference is in the fact that the second also takes a `dirfd` file descriptor, which specifies a folder, and in this case, the executable path must be interpreted as relative from this directory. Moreover, thanks to the `flags` parameter in `execveat`, it allows specifying additional behaviours like not following symbolic links or interpreting `dirfd` as a file descriptor to the target executable.

This decoder will need to register itself to the entry notification of the system calls mentioned above and extract the string parameter `pathname` and the list of arguments `argv` by iterating over them until a `NULL` pointer is encountered.
It has been decided not to analyse `dirfd` for this implementation and include it in the relative future developments Section 6.2.

In Figure 4.23, it is possible to see the final report from the Execve Decoder after tracing the following command:

```
$ /bin/sh -c 'cat /etc/passwd'
```

This is a legitimate case of using the `execve` system call used to run the `cat` command, which is an executable in `/system/bin/cat` and can be found since the `/system/bin` directory is in the `PATH` environment variable.
The output shows that in line 2, where it is possible to see that the shell has identified the executable location and in lines 4 and 5, the passed arguments are specified. By convention, the first passed argument is always the name used to invoke the executable.

```
1 ------------------ EXECVE DECODER START ------------------
2 Executable: /system/bin/cat
3 Arguments:
4 [0] = cat
5 [1] = /etc/passwd
6 ------------------ EXECVE DECODER STOP -------------------
```

Figure 4.23: Execve Decoder output showing the shell executing a command

Chapter 5

Validation

This chapter aims to validate that *Ptracer*'s tracing and analysing capabilities can be used for detecting debuggers, anomalies and privacy issues.
To do that, a demo application has been developed and used in various tests to assess how well the developed tool will perform.
Moreover, it will be shown how this will open the door for more complex analysis of complex and widespread applications.

All the tests described in the following sections have been performed on a phone model "OnePlus 6T" running Android 12.

5.1 Use case

To validate the effectiveness of *Ptracer* in detecting debuggers, anomalies and privacy issues, it is necessary to test it on an application with a well-known behaviour. This led to the development of "*AudioRecorder*", a basic Java-based application that implements a simple audio recorder which can acquire data from the microphone, save the recording and replay it.
After this first validation, it will be possible to leverage *Ptracer* to monitor more complex and well-recognised applications pursuing the same end goals.

When first started, `AudioRecorder` will show the user its interface, which is reported in Figure 5.1. In the screenshot, it is possible to see the four main buttons: start/stop recording and start/stop playing the recording.
In the latest Android versions, it is necessary to request some permissions at runtime, hence the application will check if the necessary user permissions have been granted, if not, when the record button is pressed, the typical Android prompt will appear to ask the user to grant them. More specifically, the `RECORD_AUDIO` and `WRITE_EXTERNAL_STORAGE` permissions will be requested, the first will be used to be allowed to access the microphone and the second to write on the external storage the resulting file.
The recorded data will be saved in the file `AudioRecording.3gp` stored in an application-specific directory in the external memory.

To record audio from the microphone, the application uses the Java class in the Android multimedia framework `android.media.MediaRecorder`, which includes support for capturing and encoding various common audio and video formats.

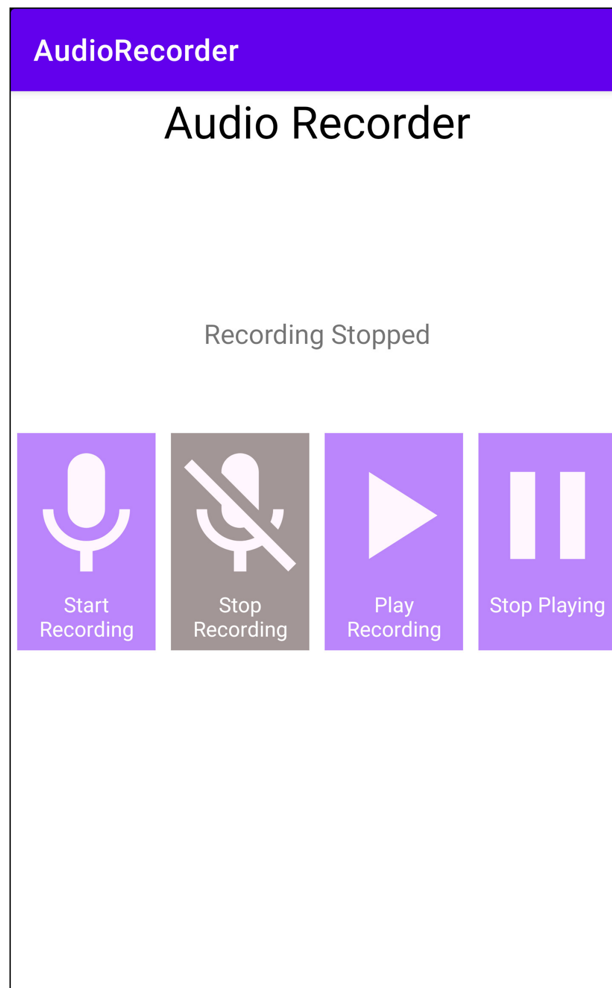


Figure 5.1: Main screen of the test application used for the validation

This class will be used after the user presses the record button, hence, a listener for this action has been created, which will invoke the method shown in Figure 5.3.
The reported source code shows the settings used to initialize `MediaRecorder`, which can be seen from lines 14 to 17 and corresponds to setting the audio input, the output file format together with the encoding and eventually the destination file.
The recording is started in line 23, which is also the method that will trigger the interaction with the Android Binder since at this point, all the necessary parameters to acquire audio are specified.

To check if the user has already been granted permission to record audio and write to the external storage, the class `androidx.core.content.ContextCompat` is used. This class is a helper to accessing features in `Context`, which is the interface to global information about an application environment.
This class allows querying the Android system to check if a specific process is allowed to do a particular action.

AudioRecorder is also able to replay the recorded audio, also in this case, `MediaRecorder` has been used since it can replay any audio file (using a supported format) given its path. This operation does not require additional permission and does not check if it can access files in the external storage since the button invoking this method is available only after a recording has been created.

```java
package it.matteodegiorgi.audiorecorder;

/* .. */

public class MainActivity extends AppCompatActivity {
  /* ... */

  private void startRecording() {
    if (CheckPermissions()) {
      /* ... Redacted buttons handling ...*/
      File dir = ContextCompat.getExternalFilesDirs(getApplicationContext(), "audio")
          [0];
      recording = new File(dir, "/AudioRecording.3gp");
      this.mRecorder = new MediaRecorder();
      this.mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
      this.mRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
      this.mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
      this.mRecorder.setOutputFile(recording);
      try {
        this.mRecorder.prepare();
      } catch (IOException e) {
        Log.e("TAG", "MediaRecorder preparation failed");
      }
      this.mRecorder.start();
      statusTV.setText("Recording Started");
    } else {
      RequestPermissions();
    }
  }

  /* ... */
}
```

Figure 5.2: Source code snippet from *AudioRecorder* showing how the recording is started

Another feature added to the basic *AudioRecorder* functionalities is the possibility to load a configuration file at startup. It contains the name of the file where recordings will be saved, and it is read and generated by using the Java native serialization on the class `Configurations`, which can be seen in Figure 5.3.

`Configurations` is a POJO class that, for simplicity, contains only one configuration and implements the `Serializable` interface, which makes it a suitable candidate to be serialized and deserialized from and to its binary format.
To do that, it is possible to leverage an `ObjectOutputStream` for serialization and a `ObjectInputStream` for deserialization, where the first writes the result into an `OutputStream`, and the second requires a `InputStream` to read a previously serialized object.
In Figure 5.4, it is possible to see the code snippet that reads the configuration file at startup.

The rationale behind the inclusion of this feature is to make the application vulnerable to an Unsafe Deserialization Vulnerability, where an attacker is able to execute arbitrary code by letting the victim application parse a malicious serialized object. In fact, when deserializing, the fully qualified class name is read from

```
package it.matteodegiorgi.audiorecorder;

import java.io.Serializable;

class Configurations implements Serializable {
    private final String recordingPath;

    public Configurations(String recordingPath) {
        this.recordingPath = recordingPath;
    }

    public String getRecordingPath() {
        return recordingPath;
    }
}
```

Figure 5.3: Source code of the `Configurations` class

```
File dir = ContextCompat.getExternalFilesDirs(getApplicationContext(), "config")[0];
File configurationFile = new File(dir, "config.bin");
Configurations configurations;
try (FileInputStream file = new FileInputStream(configurationFile);
     ObjectInputStream in = new ObjectInputStream(file)) {
    configurations = (Configurations) in.readObject();
} catch (IOException | ClassNotFoundException e) {
    Log.e("TAG", "Error occurred deserializing the configuration file", e);
}
```

Figure 5.4: Source code snippet from *AudioRecorder* showing how the configurations file is read

the object, hence before casting the result to the desired class, as can be seen in line 4, a completely different type might have been already constructed. Therefore, an attacker can craft a malicious object that, when deserialized, will run the code he desires.
To do that, the attacker can leverage all the serializable classes in the classpath and also chain them to build a complex payload, effectively using classes and method invocations as gadgets to achieve the desired behaviour.
For this thesis, the malicious object has been prepared reusing the payload named "`CommonsCollections6`" prepared by [18] via the tool `ysoserial`, which requires the dependency `commons-collections:3.1` (from the Maven Central repository) on the classpath. This dependency is very common since it is considered the recognised standard for collection handling in Java.
Leveraging this tool, it has been possible to generate a payload that invokes the `/bin/sh` shell when the application tries to deserialize the malicious object.

In an Android environment, there is an increased risk of encountering such vulnerabilities because of the inter-process communications performed via the Binder. One of the many functions of the Binder is allowing Activities and, more generically, processes to send objects among themselves. Such objects can be serialized and then transported through the IPC driver via a wrapper class (`android.content.Intent`), used as messaging object to request an action from another component.
When the object is received by the destination app, the same vulnerability might be triggered if the data source and content are not properly validated.

For simplicity, it has been decided to trigger such vulnerability just by reading a file, a more realistic scenario would be allowing other applications to send Intent instances to *AudioRecorder* containing a serialized custom class (e.g., to request a previously recording).

An application is vulnerable to such an attack when insufficient checks are made on the parsed objects, more specifically, it is necessary to ensure that the deserializer can instantiate only the intended object classes and that the data source is trusted. A potential mitigation is reducing the availability of gadgets in the classpath, but this vulnerability lies in the fact that the application parses untrusted objects and not in the availability of gadgets.

Despite *AudioRecorder* being a simple application, the fact that it is Java based and runs in an Android environment makes it very complex from the point of view of a system calls observer like *Ptracer*.
It will be observed that a large number of system calls is requested because of the event-driven nature of Android since every user interaction and communication with the system or specific service (e.g., the audio service) requires one.
Moreover, Java applications require a further level of abstraction from the operating system since the code is not directly compiled to native machine instructions but into bytecode, typically executed by the Java Virtual Machine (JVM), which can invoke additional system calls. In an Android environment, the intermediate code targets the Dalvik Executable Format (DEX) instead of the JVM; Therefore, it is fundamentally different but nevertheless still abstracted from the underlying native machine instructions. This is also valid in the most recent Android RunTime (ART), which runs applications using a hybrid approach between Ahead-Of-Time (AOT) compilation, which produces native machine code, and JIT interpretation, since there is still a middle layer handling the application.

In the next sections, this application will be analysed using *Ptracer* to validate its tracing and monitoring capabilities to detect anomalies, debuggers and privacy issues.

5.2 Anomaly detection

This section aims to evaluate the Anomaly detection capabilities of *Ptracer* and, in particular, if it can be applied to an Android application, given its particularities previously discussed.
The technique used to identify anomalies is based on an initial learning phase, where an NFA will be generated and refined over multiple learning cycles, followed by an enforcing phase, where every kernel interaction that is not expected by the model will be flagged as an anomaly, and the application halted.
The target application is going to be *AudioRecorder*, and the validation goals can be summarized in the following points:

- The learning phase converges to a stable NFA in a finite and reasonable number of iterations, i.e. the number of observed states and transitions stop increasing after a certain number of application executions.
- The enforce phase presents as few false positives as possible and only when events that have not happened during the learning phase occur since the

effectiveness of the Anomaly Detection System is as good as its training phase.

- If instead of the legitimate configurations file, a malicious object is deserialized, then the exploit payload is not executed because detected as an anomaly.

In the following sections it will be discussed the method decided to attach to the application, the learning process and eventually, the enforcing phase will be tested against the introduced vulnerability.

5.2.1 Attach to the application

When using the Linux process tracing interface, there are two main ways of attaching to a process:

1. Attach to a process that is already running, which will be stopped as a consequence of this operation;

2. Generate a child who will ask to be traced via a `ptrace` system call using the `PTRACE_TRACEME` parameter, which will stop the process until the parent successfully attaches to it. When done, it will be possible to use an `execve` system call to replace itself with the desired executable to trace.

Ptracer supports both cases, which can be specified using its command line.
When building the NFA model, it is advised to use the second approach since it guarantees to be able to observe all the generated system calls, while using the first, inevitably, some calls will go untraced. Moreover, the second approach ensures more stability in the model since it guarantees that the NFA will contain all the system calls. The first method generates a partial automaton because of the missed system calls, and it has an increased risk of finding false positives when enforced because it will be necessary to ensure that the attach operation always happens on the same automaton state.

As discussed in the related Background Section 3.3, every Android application is always launched by the Zygote process as a fork of itself. This process is always present and maintains a pool composed of its forks (called "Blastula" pool), where each preloads all the necessary libraries and Android Runtime to be ready to specialize into an individual application upon request.
This system design implies that it is not possible to use an `execve` system call to run an Android application, therefore, it is necessary to use the first tracing technique seen before.

To identify anomalies, it has been decided to start tracing the application only once its initialization is complete, hence from the moment its main activity is displayed and waiting for events.
Since Android applications are event-based, the first system call that will be encountered will always be waiting for a new event from the dedicated queue. Such behaviour will allow the establishment of a stable attach point for *Ptracer*, as it will be shown during the learning phase, with the downside of not capturing the initialization system calls.
Section 6.3 will cover the proposed future developments for this approach, and a

brand new one will be proposed to be able to trace Android applications starting from Zygote forks.

To minimize every involuntary interaction with *AudioRecorder* before attaching *Ptracer*, the application will not be started directly from the usual Android application launcher on the phone, but instead using the Activity Manager command line tool `am`. This operation is equivalent of opening the app through the usual Android application launcher directly from the phone, but it can be done remotely via a shell on the Android phone.
This is a precautionary measure since every interaction with the application would trigger an event, e.g., a touch on any part of the screen while displaying the main activity or minimizing the application, operations that would be notified as an event.
The command that will be used requires to specify the application package name and the specific activity that shall be launcher separated by a slash symbol ("/"), which is equivalent to the following:

```
$ am start -n it.matteodegiorgi.audiorecorder/\
              it.matteodegiorgi.audiorecorder.MainActivity
```

In Figure 5.5, it is possible to see a redacted output from the `pstree` command, which shows the process tree of *AudioRecorder* once started.
When launching *Ptracer*, it will be necessary to specify the process ID it should attach to. Since all of the source code developed for the application runs in the principal application process (which in the example can be seen in line 1 with PID 5663), it has been decided to trace only this one.
The rationale behind this choice is that *AudioRecorder* is a very simple application, and all the other observed processes do not have a role in its main logic.
Tracing more complex applications, which effectively leverage multiple processes and threads, will be covered in the future developments Section 6.4.

```
1  init(1)-+-main(858)-+-i.audiorecorder(5663)-+-{ADB-JDWP Connec}(9589)
2          |           |                        |-{Binder:5663_1}(9607)
3          |           |                        |-{Binder:5663_2}(9610)
4          |           |                        |-{Binder:5663_2}(9625)
5          |           |                        |-{Binder:5663_3}(9611)
6          |           |                        |-{FinalizerDaemon}(9603)
7          |           |                        |-{FinalizerWatchd}(9606)
8          |           |                        |-{HeapTaskDaemon}(9601)
9          |           |                        |-{Jit thread pool}(9591)
10         |           |                        |-{Profile Saver}(9618)
11         |           |                        |-{ReferenceQueueD}(9602)
12         |           |                        |-{RenderThread}(9619)
13         |           |                        |-{Signal Catcher}(9587)
14         |           |                        |-{hwuiTask0}(9623)
15         |           |                        |-{hwuiTask1}(9624)
16         |           |                        \-{perfetto_hprof_}(9588)
17 ...
```

Figure 5.5: Process tree of *AudioRecorder*

Moreover, it has been noticed that the Android RunTime continuous code profiling and optimization can pose another factor of instability for the acquired stack traces. In fact, ART, by default, will analyse the executed code and record profiling samples that will be used to determine which code sections can be considered "hot" or not. Hot code consists of frequently used methods which will be compiled into machine code by the Ahead Of Time (AOT) compiler when identified as such.
This implies that over multiple runs, the executed code will evolve from being

completely Just In Time interpreted to a mix of interpreted bytecode and pre-compiled (hence, machine code). Moreover, thanks to the On-Stack Replacement (OSR) feature of ART, it is possible to replace JIT code right after it gets compiled and at run-time.
Such replacements will cause stack traces not to be reproducible until ART has reached a stable equilibrium between interpreted and compiled code, which might depend on the specific user's behaviour. This is because between compiled and interpreted code, there are bridging stack frames that will be inserted, and if they are not always in the same position, then an Unknown State Anomaly will be generated.
Hence, handling both machine and JIT code is not a problem for *Ptracer*, but the constantly changing balance between these parts would make the learning process harder.

For these reasons, to validate *Ptracer*'s capability to identify anomalies, it has been decided to remove this uncertainty factor by disabling AOT compilation for the tested application.
This can be done by modifying the `AndroidManifest.xml` file by including the following parameter in the `application` XML element: `android:vmSafeMode="true"`.
To ensure that the previously compiled code is not used, the related AOT code cache has been deleted using the following command run via a shell on the Android phone:

```
$ cmd package compile --reset it.matteodegiorgi.audiorecorder
```

Such change has greatly improved the stability of stack traces, also after multiple executions, since now the application code is not adaptively compiled.
It has been possible to validate that no application code is getting compiled by observing that the stack frames to transition from JIT to AOT are now always in fixed positions. Hence, after multiple executions, it is faster to reach the model convergence, and there are no new states which differ from a previously observed one just for the bridging stack frames.

Alternatively, it would also be possible to trigger a complete application compilation by the AOT compiler since, from the point of view of the Anomaly Detector, it is just desired that the produced stack frames are as reproducible as possible. This can be done using the following command:

```
$ cmd package compile -m speed -f it.matteodegiorgi.audiorecorder
```

Now it will be necessary to disable the JIT interpreter since if both JIT and AOT code exists, then the first is preferred. This can be done by running the following command, which sets the related system property:

```
$ setprop dalvik.vm.usejit false
```

From the performed experimentation, this method has provided less stable results. Hence, it has been chosen not to use AOT code to detect anomalies, also because the generated bytecode is less machine-specific than the machine code generated by this approach, and this would make the learnt model more portable among multiple devices.

5.2.2 Learn the model

The detection accuracy and effectiveness of *Ptracer* as an Anomaly Detection System (ADS) are directly linked with the training quality of its model of normal behaviour since it identifies as attacks any divergence from normal behaviour. Hence, it is first necessary to have a model of what can be considered "normal" to be then able to enforce it and raise alerts. This type of detection can detect any unexpected behaviour, which must be analyzed and classified as malicious or an application malfunction.

The objective of this section is to learn the NFA model and iteratively refine it by running the application and operating it. As discussed in the previous section, the target application is going to be *AudioRecorder*, and *Ptracer* is going to attach to it following the procedure previously discussed.

To assess how fast the model will converge, it will be measured how many learning iterations will be needed to obtain a steady number of NFA states, transitions and final states. In fact, it is necessary to ensure that the model is as stable as possible during learning to minimize the number of false positives during the subsequent enforcing phase.
The desired outcome is given by extensive learning, where all the possible user behaviours are covered.

It is now possible to start the learning process by launching the application, retrieving the PID of the main process and running *Ptracer*. The application launch has been described in the previous section, and the tracing program can be started using the following command:

```
./ptracer --decoders false \                        # Disable system call decoders
          --authorizer true \                       # Enable the Authorizer component
          --learn true \                            # Set learning mode
          --nfa audiorecorder.nfa \                 # NFA model file
          --associations audiorecorder.ass \        # Mapper associations file
          --dot audiorecorder.dot \                 # NFA DOT representation file
          --name it.matteodegiorgi.audiorecorder \ # Name of the trace application
          --pid $(ps -A | grep matteo | awk '{ print $2;}') # Find the PID to trace
```

The first option disables the System Call Decoders (described in Section 4.5) since they would slow down the overall process and are not used in this instance.
The other parameters are used to enable the Authorizer component, which contains the logic to learn and enforce the NFA model. Its configurations include activating learning mode and specifying the files where are the NFA model and associations file (which contains the associations created by the *Mapper* function, explained in Section 4.4), which will be imported (if they already exist) or saved when the traced application terminates.
The shown command also shows how to generate the DOT representation of the learnt NFA, this format can be used to visualize the generated automata later, and it is optional.
The last necessary parameters are the PID to trace and the executable name, and the last is used to set the application's package name, which is necessary because the tracer always needs to place this information in the 3-tuples associated with the NFA states. The Process ID is not directly specified in the provided command, but it is fetched using the reported concatenation of commands, which will list all the processes, look for *AudioRecorder*, isolate its PID and place it in the command

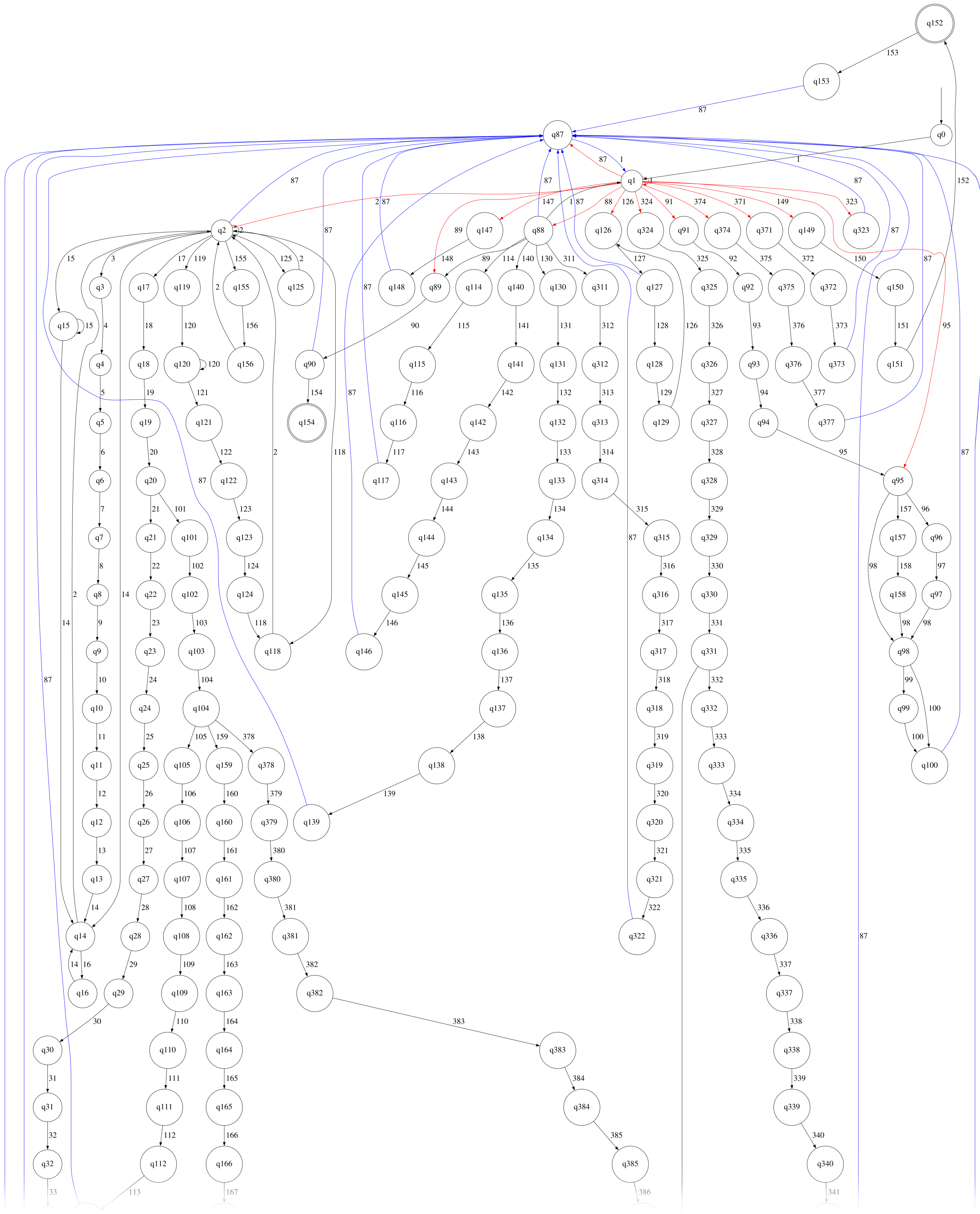


Figure 5.6: Partial representation of the NFA model obtained after the first learning iteration on *AudioRecorder*

that will launch *Ptracer*.
More information about how to run *Ptracer* and its parameters will be provided in Appendix A.

Once the application is started and *Ptracer* attached, it is necessary to simulate the normal application usage. For the first learning iteration, it has been decided to do the following: start/stop the recording and replay and pause it. Eventually, the application was closed by swiping it up from the list of recent applications on the Android phone.
This simple interaction has generated 378 unique states (including `q0`, the initial state), with 418 transitions among them and 3 states marked as final.

In Figure 5.6, it is possible to see the graphical representation of the NFA obtained after the first learning iteration, it has been necessary to crop it given its large size.
The NFA structure is a direct reflection of the application's internal structure, which is always waiting for events coming from a queue and dispatching them to multiple listeners that have registered their interest in some specific type of event.
In the NFA representation, the state `q0` is the entry point, which is pointed by an arrow without any source in the top right area of the automaton. This state is not associated with any 3-tuple mapped by *Mapper*, but it represents the attach operation performed by *Ptracer*, which, in this case, has immediately encountered the system call `epoll_pwait`, identified by the transition 1 leading to `q1`.
Looking into the list of produced associations, it is possible to see that state 1 has the following method call in its stack trace:

```
android::android_os_MessageQueue_nativePollOnce(_JNIEnv*, _jobject*, long, int)@44
```

Hence, it is reasonable to assume that when the tracer attached to the application, it was waiting for new events coming from the event queue. In fact, starting from `q1`, it is possible to see multiple outgoing transitions, which lead to the various event handlers and have been manually coloured in red in the reported automaton.
Moreover, there is a recurring pattern among all the handlers since, every time they complete their operations, they lead back to state `q87`, all such transitions have been coloured in blue. From the extracted stack trace for this state, it is possible to see that it corresponds to a call to the following method, which is part of the final clean-up after an event is handled:

```
android::IPCThreadState::clearCallingIdentity()@40
```

Therefore, the action sequence used during the various tests is not inherently reflected in the model if the generated events are not logically dependent among them, e.g., a touch on a blank part of the screen will not influence any of the button-pressed events, but if the recording button is pressed twice in a row, then the second event will be influenced by the first since it will do nothing. This is a significant property since it ties the model to the logic that developers implemented and not to the action sequences performed during testing, which will only need to guarantee that all the possible combinations are covered.

Eventually, the states marked as final are all of those states where it has been observed the termination of any of the traced threads or processes. In this case, they are 3, where two can be seen from the partially reported automaton (`q152` and `q154`), and one is further away and has not been shown (`q258`).

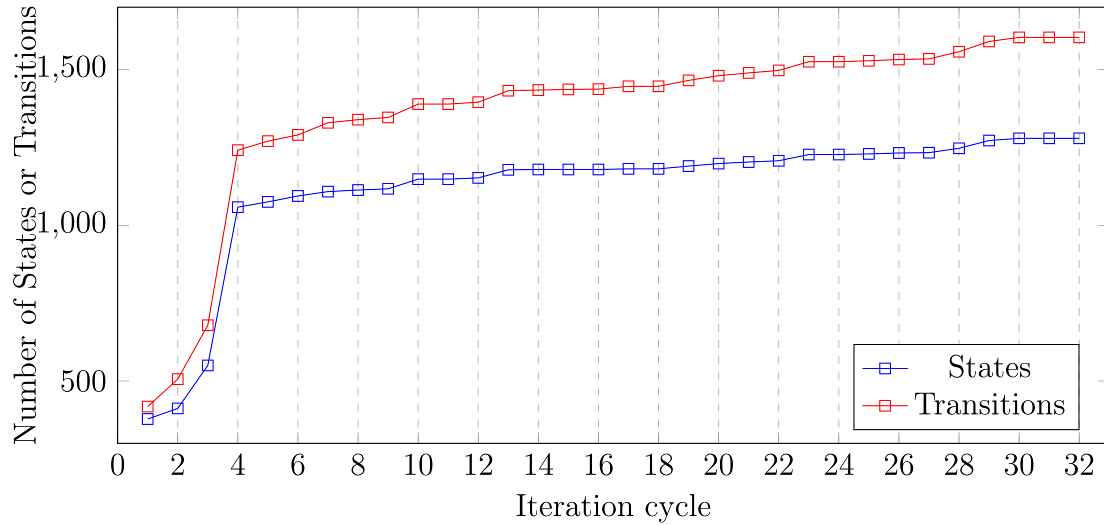
The reason why there are multiple termination states is that the application is actually multithreaded, in fact, states `q331` (visible in the figure) and `q227` (not visible) corresponds to `clone` system calls. These calls are triggered by the audio media player invoked by pressing the play recording button.

Iteration	States	Transitions	Final States	Included Event Types
1	378	418	3	Only button usage
2	412	506	4	Random display events
3	550	679	5	Minimize and restore
4	1058	1241	5	Minimize and reopen
5	1075	1270	6	Activation of notification bar
6	1094	1290	7	Same as above
7	1108	1329	8	Same as above
8	1113	1339	8	Same as above
9	1117	1346	8	Same as above
10	1148	1389	9	Same as above
11	1148	1389	9	Same as above
12	1152	1395	10	Same as above
13	1178	1432	10	Same as above
14	1179	1434	10	Same as above
15	1179	1436	11	Same as above
16	1179	1437	12	Same as above
17	1181	1446	13	Same as above
18	1181	1446	13	Same as above
19	1190	1465	13	Same as above
20	1198	1480	13	Same as above
21	1203	1489	13	Same as above
22	1207	1497	13	Same as above
23	1227	1525	13	Same as above
24	1227	1525	13	Same as above
25	1229	1528	13	Same as above
26	1232	1532	13	Same as above
27	1233	1534	14	Same as above
28	1247	1557	14	Same as above
29	1272	1590	14	Same as above
30	1274	1594	14	Same as above
31	1274	1594	14	Same as above
32	1274	1594	14	Same as above

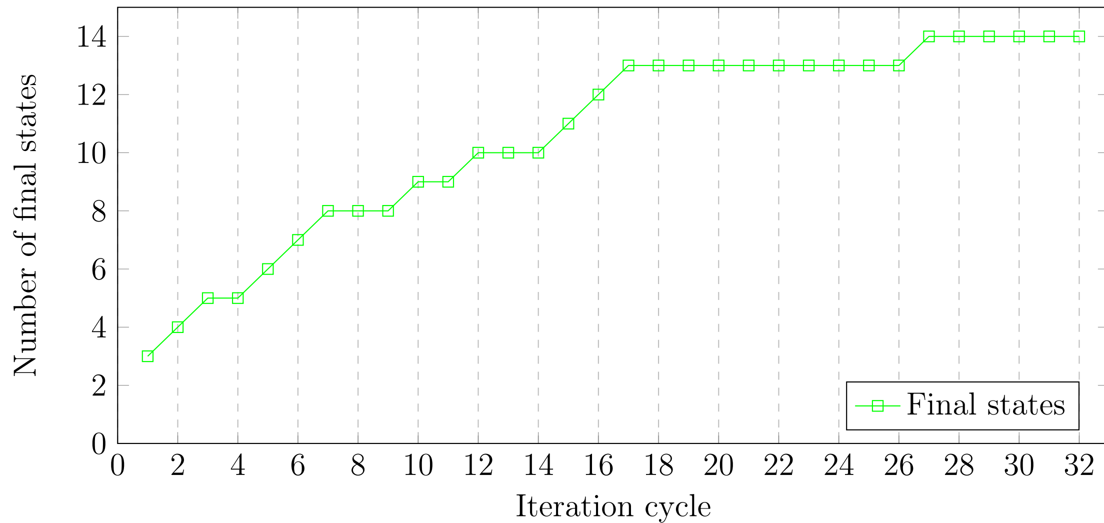
Table 5.1: Table reporting the NFA model data per each learning iteration

After the first learning iteration, it was necessary to re-run the application multiple times and add different user action sequences and combinations. For example, in the second learning iteration, also touch events in blank parts of the screen (hence, not handled by any button) were included, and this already implied a significative increment of the NFA model.
Since the automaton complexity increased every iteration, it has been possible to graphically report only the result of the first iteration since the other ones would be too big and complex for this document. For this reason, the generation of the

NFA representation in a DOT file format has been disabled after the first iteration.



(a) Plot showing the total number of NFA states and transitions per learning cycle



(b) Plot showing the total number of final NFA states per learning cycle

Figure 5.7: Plots the NFA model status during the various learning iterations

In Table 5.1, it is possible to see a summary of how the NFA model evolved during the various learning iterations. It has been decided to incrementally include events to be able to observe the various growths in the automaton. Hence, each iteration contains all the actions performed in the previous ones plus the newly included events.
In iteration 3, it was included the application minimization and restoration via the "Recent Applications" menu. This has been expanded in iteration 4, where the application was minimized and reopened via the activity launcher, this triggered the recreation of the activity from scratch, which is explained by the significant growth of the model. The last included event was swiping down the notification bar.

In Figure 5.7a and Figure 5.7b, it is possible to see plotted the growth trend of the NFA, more specifically the first plot shows transitions and NFA states, and the second the final states.

From the reported plots, it is possible to see how, after the first three iterations, the model starts to converge to its stability. For this test, the model has been considered stable only if, after three learning cycles, the automaton did not grow. This condition happens only from the 30th learning iteration; hence the last iteration was the 32nd, where the model has finally been considered stable.

The final stable model has 1279 states, 1603 transitions and 14 final states, its enforcement will be discussed in the next section, where it will be tested and evaluated.

5.2.3 Enforcement phase

The desired outcome from this section is assessing the number of false positives, which are anomalies during the normal application execution, and evaluating whether the learning phase performed in the previous section produced a satisfactory model.
Moreover, the model will be tested to determine whether it will be able to prevent the execution of arbitrary code triggered by an unsafe deserialization attack vector. These vulnerabilities have been better described in Section 5.1.

To identify potential false positives, *AudioDecoder* has been started via the phone activity launcher (instead of using the `am` command line utility used during learning), and *Ptracer* attached using the following command:

```
./ptracer --decoders false \                      # Disable system call decoders
          --authorizer true \                     # Enable the Authorizer component
          --learn false \                         # Set enforce mode
          --nfa audiorecorder.nfa \               # NFA model file
          --associations audiorecorder.ass \      # Mapper associations file
          --name it.matteodegiorgi.audiorecorder \ # Name of the trace application
          --pid $(ps -A | grep matteo | awk '{ print $2;}') # Find the PID to trace
```

The main difference between the learning and enforcement phase is the deactivation of the `--learn` option since all the other parameters are still needed (except `--dot`, which has only a graphical function).

To identify potential false positives, it has been decided to follow the same action order followed during the learning phase. Therefore, first, the buttons will be used, then other display events will be included, and eventually, the app minimization and restoration or reopening will be performed.

The NFA model proved well-trained in the normal execution of start/stop recording and replaying the audio file. Running this action sequence and its various combinations have not triggered any anomaly after 10 complete application executions. Hence, this also validates that attaching to the application while it is waiting for an event consists of a stable attachment point.
When the range of actions was extended by including also display events, touching blank parts of the screen, and the application minimization and restoration, no anomalies were registered.
This is a very satisfactory result, as it shows that the learnt model is a good fit for the normal application usage.

After eight executions of *AudioRecorder*, a false positive unknown state anomaly occurred while restoring an application from its minimized state. This anomaly

can be seen in Figure 5.8, while its stack trace has been partially redacted for the sake of clarity.

```
------------------ SYSCALL ENTRY START ------------------
Notification origin: it.matteodegiorgi.audiorecorder
PID: 18983
SPID: 18983
Timestamp: 1676730433865611
NOT Authorized
Syscall = futex (98)
Stack unwinding = {
  PC 0x0000007c4d0a3260 Relative PC 0x000000000004c260 SP 000000007ffb554520 - syscall @ 32
  PC 0x0000007c4d109b48 Relative PC 0x00000000000b2b48 SP 000000007ffb554520 - pthread_mutex_unlock @ 112
  PC 0x0000007c40738a38 Relative PC 0x00000000000eea38 SP 000000007ffb554550 - android::Singleton<android::
       ComposerService>::getInstance() @ 112
  PC 0x0000007c40738ed4 Relative PC 0x00000000000eeed4 SP 000000007ffb554580 - android::ComposerService::
       getComposerService() @ 36
  PC 0x0000007c407414f0 Relative PC 0x00000000000f74f0 SP 000000007ffb5545c0 - android::SurfaceComposerClient
       ::onFirstRef() @ 48
  PC 0x0000007c40738668 Relative PC 0x00000000000ee668 SP 000000007ffb554620 - android::SurfaceControl::
       readFromParcel(android::Parcel const&, android::sp<android::SurfaceControl>*) @ 724
  PC 0x0000007c27464390 Relative PC 0x000000000012e390 SP 000000007ffb5546c0 - android::nativeReadFromParcel(
       _JNIEnv*, _jclass*, _jobject*) @ 56
  ...
  PC 0x0000007c273f6fb4 Relative PC 0x00000000000c0fb4 SP 000000007ffb557ba0 - android::AndroidRuntime::start
       (char const*, android::Vector<android::String8> const&, bool) @ 836
  PC 0x00000055e36d258c Relative PC 0x000000000000258c SP 000000007ffb557c90 - main @ 1336
  PC 0x0000007c4d09f7dc Relative PC 0x00000000000487dc SP 000000007ffb558e00 - __libc_init @ 96
}
Parameters = {
  0x0000007c4077f760
  0x0000000000000081
  0x0000000000000001
  000000000000000000
  000000000000000000
  000000000000000000
  000000000000000000
  000000000000000000
}
Registers = {
  PC: 0x0000007c4d0a3260
  SP: 0x0000007ffb554520
  RET: 0x0000007c4077f760
}
------------------ SYSCALL ENTRY STOP ------------------
State not found in the list of associations -> Not authorised
Warning! Encountered a state that has never been observed before!

Possible actions:
1 - Kill the target process
2 - Allow it by adding a new state in the model
Choice: 2
Added a new transition from q495 to q1247
...
```

Figure 5.8: Unknown State Anomaly found while restoring the application

It was possible to recover from this anomaly by choosing to add a new state (`q1247` since they are numbered starting from 0) and a transition to it. Alternatively, in case the observed behaviour is too suspicious, it would have been possible to decide to terminate the traced application and leave the automaton unaltered.
Anomalies rarely involve only one state, in fact, it was necessary to repeat the same operation for the next 3 states before going back to a previously known path inside the automaton.
When this execution path, which was missed in the learning phase, was added, all the subsequent executions manage to restore the application from its minimized state without further anomalies.

```
1  ------------------ SYSCALL ENTRY START ------------------
2  Notification origin: it.matteodegiorgi.audiorecorder
3  PID: 18984
4  SPID: 18984
5  Timestamp: 1676731779196358
6  NOT Authorized
7  Syscall = clock_gettime (113)
8  Stack unwinding = {
9    PC 0x0000007c5192a618 Relative PC 0x0000000000000618 SP 000000007ffb552020 - __kernel_clock_gettime @ 680
10   PC 0x0000007c4d0a2438 Relative PC 0x000000000004b438 SP 000000007ffb552020 - clock_gettime @ 24
11   PC 0x0000007990473fb0 Relative PC 0x0000000000673fb0 SP 000000007ffb552040 - art::Thread::GetCpuMicroTime()
         const @ 56
12   PC 0x000000799023fc10 Relative PC 0x000000000043fc10 SP 000000007ffb552080 - art::(anonymous namespace)::
        ScopedCheck::CheckPossibleHeapValue(art::ScopedObjectAccess&, char, art::(anonymous namespace)::
        JniValueType) @ 2944
13   PC 0x000000799023e8ac Relative PC 0x000000000043e8ac SP 000000007ffb552140 - art::(anonymous namespace)::
        ScopedCheck::Check(art::ScopedObjectAccess&, bool, char const*, art::(anonymous namespace)::
        JniValueType*) @ 644
14   PC 0x000000799023b45c Relative PC 0x000000000043b45c SP 000000007ffb5521e0 - art::(anonymous namespace)::
        CheckJNI::GetPrimitiveArrayCritical(_JNIEnv*, _jarray*, unsigned char*) (.llvm.11011294371123972320)
        @ 664
15   ...
16   PC 0x0000007c273f6fb4 Relative PC 0x00000000000c0fb4 SP 000000007ffb557ba0 - android::AndroidRuntime::start
        (char const*, android::Vector<android::String8> const&, bool) @ 836
17   PC 0x00000055e36d258c Relative PC 0x000000000000258c SP 000000007ffb557c90 - main @ 1336
18   PC 0x0000007c4d09f7dc Relative PC 0x00000000000487dc SP 000000007ffb558e00 - __libc_init @ 96
19 }
20 Parameters = {
21   0x00000000fffdaebe
22   0x0000007ffb552040
23   0x0000000000000045
24   0x0000007a53e54b90
25   0x0000007ffb552240
26   000000000000000000
27   000000000000000000
28   000000000000000000
29 }
30 Registers = {
31   PC: 0x0000007c5192a618
32   SP: 0x0000007ffb552020
33   RET: 0x00000000fffdaebe
34 }
35 ------------------ SYSCALL ENTRY STOP ------------------
36 There are no possible transitions from q2 to q1034
37 System call NOT authorised
38 Warning! Encountered a transition that has never been observed before!
39
40 Possible actions:
41 1 - Kill the target process
42 2 - Allow it by adding a new transition in the model
43 Choice: 2
44 Added a new transition from q2 to q1034
```

Figure 5.9: Unknown Transition Anomaly found while reopening the application

While testing the recreation of the application's main activity, it was discovered that it is triggered only when the application is started via the `am` command line utility, then minimized and reopened using the Android activity launcher on the phone (hence, clicking on the application icon). Therefore, it was necessary to use `am` to trigger it and not only the Android application launcher as done for the previous tests.
While testing the model quality during the recreation of the main activity, plenty of anomalies were encountered. One of them can be seen in Figure 5.9, it is an unknown transition anomaly and, in this case, *Ptracer* asks whether it is desired to add a new transition or kill the application.
It was decided to add it to the automaton and proceed, then another 10 unknown state anomalies followed and as many states were added to the automaton.

In this case, the quality of the trained model is not good enough. Looking at the produced anomalies, it is possible to hypothesize that during the application start-up, many synchronization operations on mutexes need to occur, and in this case, it is very hard to be able to capture all the possible combinations in the model.
In the future development Section 6.5, it will be discussed how to improve the model to be able to be more flexible in these cases.

```
1  ----------------- SYSCALL ENTRY START ------------------
2  Notification origin: it.matteodegiorgi.audiorecorder
3  PID: 28738
4  SPID: 28738
5  Timestamp: 1676751181776975
6  NOT Authorized
7  Syscall = read (63)
8  Stack unwinding = {
9   PC 0x0000007c4d0f4238 Relative PC 0x000000000009d238 SP 000000007ffb5527e0 - read @ 8
10  PC 0x0000007986469b6c Relative PC 0x0000000000029b6c SP 000000007ffb5527e0 - Linux_readBytes(_JNIEnv*,
        _jobject*, _jobject*, _jobject*, int, int) @ 172
11  ...
12  PC 0x000000798fa76e00 Relative PC 0x00000000000bde00 SP 000000007ffb553ac0 - java.io.
        ObjectInputStream$BlockDataInputStream.readBlockHeader @ 56
13  ...
14  PC 0x000000798fa77cd8 Relative PC 0x00000000000becd8 SP 000000007ffb553e20 - java.io.
        ObjectInputStream$BlockDataInputStream.skipBlockData @ 16
15  ...
16  PC 0x0000007c405522de Relative PC 0x00000000000012de SP 000000007ffb5549b0 - it.matteodegiorgi.
        audiorecorder.MainActivity.startRecording @ 78
17  ..
18  PC 0x0000007c273f6fb4 Relative PC 0x00000000000c0fb4 SP 000000007ffb557ba0 - android::AndroidRuntime::start
        (char const*, android::Vector<android::String8> const&, bool) @ 836
19  PC 0x00000055e36d258c Relative PC 0x000000000000258c SP 000000007ffb557c90 - main @ 1336
20  PC 0x0000007c4d09f7dc Relative PC 0x00000000000487dc SP 000000007ffb558e00 - __libc_init @ 96
21 }
22 Parameters = {
23  0x0000000000000060
24  0x00000079a3e636f0
25  0x0000000000000001
26  0xffffffffffffffff
27  0xffffffffffffffff
28  0x0000000000000049
29  0x284801ff3a445328
30  000000000000000000
31 }
32 Registers = {
33  PC: 0x0000007c4d0f4238
34  SP: 0x0000007ffb5527e0
35  RET: 0x0000000000000060
36 }
37 ----------------- SYSCALL ENTRY STOP ------------------
38 There are no possible transitions from q57 to q64
39 System call NOT authorised
40 Warning! Encountered a transition that has never been observed before!
41
42 Possible actions:
43 1 - Kill the target process
44 2 - Allow it by adding a new transition in the model
45 Choice:
```

Figure 5.10: Anomaly found when trying to exploit the unsafe deserialization vulnerability

The overall result can be considered positive since, during the normal operations performed inside the application, no anomalies were identified.
Two false positive anomalies were identified while interacting with Android features, like restoring the app from the background or recreating its main activity. These two activities proved to be very system-call intensive, which has, as a con-

sequence, an increased chance of raising anomalies.

It is now possible to test whether the learnt model could prevent the unsafe deserialization vulnerability from being exploited.
The configuration file is read when the recording button is pressed for the first time. Hence, it will be enough to swap the configuration file `config.bin` located in the following path:
`/sdcard/Android/data/it.matteodegiorgi.audiorecorder/files/config/`
with its previously generated malicious version, which will try to execute the `/bin/sh` executable.
Ptracer successfully managed to halt the application execution and detected an anomaly that can be seen in Figure 5.10.
In this case, the anomaly was detected before the exploit payload was reached, and it was triggered by the fact that the structure of the new configuration object is very different from the expected one. Therefore, since now there is a more complex structure to parse, which also contains nested complex objects, different method calls in different orders are needed.
If it is decided to continue, it will be possible to see many unknown state anomalies given by the multiple reads invoked by the various parsers for specific data types never used in the legitimate configuration file. Eventually, the execution would lead to a state like the one in Figure 5.11, which would surely be identified as an anomaly, but before then, plenty more would be raised.

`Ptracer` has demonstrated to be effective in detecting the exploitation of this kind of vulnerability, and more generically, it is effective in all those cases where Remote Code Execution capabilities are achieved given its capability to describe the program behaviour in such a strict way.
Despite the fact that satisfactory results have been achieved in this area, it is necessary to note that the used anomaly detection mechanism would not be effective against all kinds of vulnerabilities. For example, logic vulnerabilities could go unnoticed since they keep the program on its legitimate track but exploit flows in the application design and implementation. Only limited effectiveness against denial of service attacks is possible, they would be detected only if unusual errors are triggered or the program terminates in a non-final state.

```
------------------ SYSCALL ENTRY START ------------------
Notification origin: it.matteodegiorgi.audiorecorder
PID: 4028
SPID: 4028
Timestamp: 1676754300545058
NOT Authorized
Syscall = execve (221)
Stack unwinding = {
  PC 0x0000007c4d0f3d98 Relative PC 0x000000000009cd98 SP 000000007ffb54f020 - execve @ 8
  PC 0x0000007c4d0a9b38 Relative PC 0x0000000000052b38 SP 000000007ffb54f020 - execvpe @ 96
  PC 0x0000007985c330e4 Relative PC 0x000000000002c0e4 SP 000000007ffb54f0b0 - startChild @ 936
  PC 0x0000007985c329ec Relative PC 0x000000000002b9ec SP 000000007ffb550140 - UNIXProcess_forkAndExec @ 784
  ....
  PC 0x000000798fa9823a Relative PC 0x00000000000df23a SP 000000007ffb550fc0 - java.lang.Runtime.exec @ 2
  ...
  PC 0x000000791bd5d426 Relative PC 0x0000000000285426 SP 000000007ffb551ca0 - org.apache.commons.collections
       .functors.InvokerTransformer.transform @ 46
  ...
  PC 0x000000791bd5c322 Relative PC 0x0000000000284322 SP 000000007ffb552100 - org.apache.commons.collections
       .functors.ChainedTransformer.transform @ 18
  ...
  PC 0x000000791bd6c510 Relative PC 0x0000000000294510 SP 000000007ffb552540 - org.apache.commons.collections
       .map.LazyMap.get @ 20
  ...
  PC 0x000000791bd62444 Relative PC 0x000000000028a444 SP 000000007ffb552970 - org.apache.commons.collections
       .keyvalue.TiedMapEntry.getValue @ 8
  ...
  PC 0x000000791bd623d8 Relative PC 0x000000000028a3d8 SP 000000007ffb552da0 - org.apache.commons.collections
       .keyvalue.TiedMapEntry.hashCode @ 0
  ...
  PC 0x000000798fb67fe4 Relative PC 0x00000000001aefe4 SP 000000007ffb553150 - java.util.HashMap.hash @ 8
  ...
  PC 0x000000798fb68618 Relative PC 0x00000000001af618 SP 000000007ffb5532e0 - java.util.HashMap.put @ 0
  ...
  PC 0x000000798fb6952c Relative PC 0x00000000001b052c SP 000000007ffb5534b0 - java.util.HashSet.readObject @
        152
  ...
  PC 0x000000798fa7fe46 Relative PC 0x00000000000c6e46 SP 000000007ffb554130 - java.io.ObjectStreamClass.
       invokeReadObject @ 26
  ...
  PC 0x000000798fa7a494 Relative PC 0x00000000000c1494 SP 000000007ffb5542e0 - java.io.ObjectInputStream.
       readSerialData @ 116
  ...
  PC 0x000000798fa79bb6 Relative PC 0x00000000000c0bb6 SP 000000007ffb5544a0 - java.io.ObjectInputStream.
       readOrdinaryObject @ 174
  ...
  PC 0x000000798fa79996 Relative PC 0x00000000000c0996 SP 000000007ffb554650 - java.io.ObjectInputStream.
       readObject0 @ 406
  ...
  PC 0x000000798fa79760 Relative PC 0x00000000000c0760 SP 000000007ffb554800 - java.io.ObjectInputStream.
       readObject @ 24
  ...
  PC 0x0000007c405522de Relative PC 0x00000000000012de SP 000000007ffb5549b0 - it.matteodegiorgi.
       audiorecorder.MainActivity.startRecording @ 78
  ...
  PC 0x0000007c273f6fb4 Relative PC 0x00000000000c0fb4 SP 000000007ffb557ba0 - android::AndroidRuntime::start
       (char const*, android::Vector<android::String8> const&, bool) @ 836
  PC 0x00000055e36d258c Relative PC 0x000000000000258c SP 000000007ffb557c90 - main @ 1336
  PC 0x0000007c4d09f7dc Relative PC 0x00000000000487dc SP 000000007ffb558e00 - __libc_init @ 96
}
Parameters = {
  0x00000079a3e5dbf0
  0x00000079b3ead3c0
  0x0000007ffb558ea0
  0x0000000000000020
  0x00000000c0300c03
  0xa8008101aaaa0aa0
  0xffffffffffffffff
  000000000000000000
}
Registers = {
  PC: 0x0000007c4d0f3d98
  SP: 0x0000007ffb54f020
  RET: 0x00000079a3e5dbf0
}
------------------ SYSCALL ENTRY STOP ------------------
```

Figure 5.11: Execve state triggered when the unsafe deserialization exploit is used

5.3 Debugger detection

This section wants to evaluate how suitable is *Ptracer* in being used for the purpose of detecting debuggers.
In an Android environment, applications are often written either in Java (or alternatives able to produce compatible bytecode) or in languages that can generate machine code directly, e.g., C or C++. These two categories are approached very differently by a MATE attacker since the toolset used to get a deeper understanding of the code are different.
Debuggers are one of the most used and useful tool categories for these attach types, given their possibility to halt the code and inspect and alter its internal proprieties (e.g., variable values). For example, an attacker could use a debugger to view the data transmitted to a remote server before it gets encrypted or alter a game's dynamics to gain an unfair advantage.

There are various types of debuggers, and they are implemented differently based on the languages that they target. In the case of an interpreted language, they will connect to the interpreter and drive the code execution from there, and in the case of compiled languages, they will use the Linux Process Tracing interface, hence the `ptrace` system call. For example, a Java debugger would setup the interpreter (e.g., the JVM or ART) to allow itself to connect to it and control the application, instead, a C++ debugger would use `ptrace` to directly control the target process and inspect or alter its memory.
In this section, both cases will be investigated, and various ways to detect the usage of debuggers will be covered.

To detect debuggers, there are various possible strategies, for example, the ones mentioned in the OWASP MASTG [5]. Some of them aim to prevent the attacker from being able to attach a debugger in various ways, for example, by tampering with the data structures used by a debugger or using `ptrace` to attach to themselves, therefore, excluding other tracers (since only one per process is admitted). Other techniques involve detecting whether a debugger is attached, for example, reading the process status (in `/proc/self/status`) and checking if the field "TracerPid" is not 0.
All these techniques can be easily bypassed when an attacker has full control of the application from its start since he can prevent it from tampering with some data and alter the return values of methods and system calls.

One of the most effective techniques for detecting the presence of a debugger is using time checks, leveraging on the fact that a debugger would inevitably slow down the process. To evade this detection mechanism, an attacker must tamper with all the system calls that return the current time and provide values coherent with the application model. An operation that is extremely time-consuming and, in the case of a client-server application, this process might happen on the server, which would not clearly inform the application about a detected tampering attempt.

To prevent debuggers from attaching to an application via `ptrace`, one of the most effective techniques is embedding a tracer in the application, which will not only be attached to the main processes, preventing other tracers from doing the same, but also actively have a role in the application logic. Hence, introducing a strong dependency among these two components that is difficult to break from an

attacker's point of view.
A similar example can be seen in [1], where the code fragments are transferred between the tracer and processes, making the application useless without its tracer.

The approach investigated in this section is specialized for client-server applications, where there is already a strong dependency between client and server. Such dependence will be relied on to provide tracing data to the server as a "price" to pay for the application to be considered genuine and remain part of the network of the authorized devices.
Therefore, a MATE attacker would not be able to simply kill *Ptracer* to be able to attach to the application since the server would stop receiving tracing data and act accordingly. The server response could vary, and it is advised to give the client as few clues as possible when identified as tampered with.

An attacker could run two instances of the application simultaneously, one legitimately used while the second being debugged, and then feed to the server the same tracing data for both.
For this reason, it is always advised to use multiple layers of protection against MATE attacks, for example, challenging every client to execute diversified mobile code challenges and expecting a coherent result in *Ptracer*'s data. This approach implies sending a different challenge to execute to each client, which could perform remote attestation (e.g., sending hashes of memory areas to the server) and trigger specific sequences of system calls that are unique per client.

5.3.1 Architecture

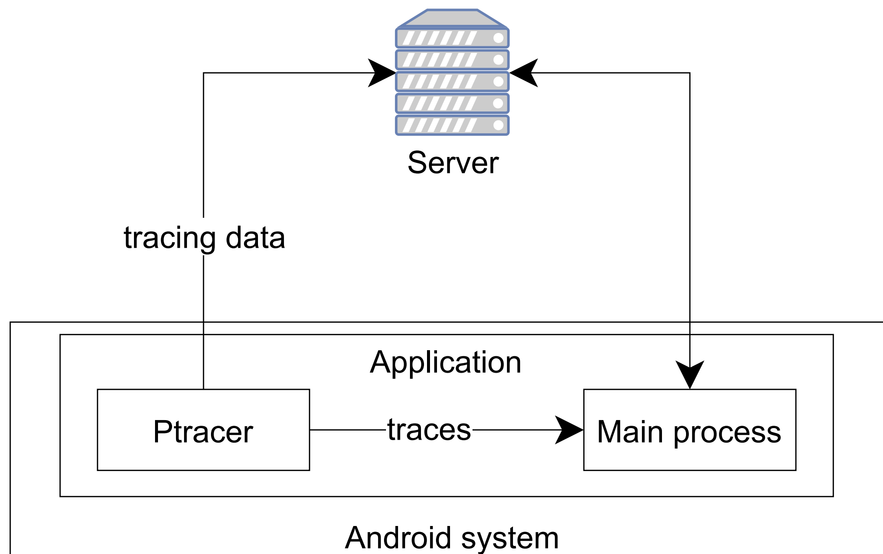


Figure 5.12: Architecture used for detecting debuggers

A diagram of the proposed architecture to detect debuggers can be seen in Figure 5.12 and consists of a client application that needs to be protected, a remote server which is interested in knowing if a client is being tampered with and the monitoring tool: *Ptracer*.
In this case, *Ptracer* is embedded in the target application, placing it inside its APK since Android strictly enforces a W^X policy, i.e. if a file is in a writable location, then it cannot be executed.

The data collected by the tracer will be sent to the server, which will analyse them and make a decision whether the client is still genuine or not. This decision considers the timings between system calls and if the execution path taken by the application is conformant with the previously learnt NFA model.

Therefore, in this case, *Ptracer*'s architecture (previously seen in Figure 4.1) is split into two machines, where the Tracer component will acquire data on the device, and the Analyzer thread will run on the server.
These two elements are already separated in the original architecture, hence this change can be implemented by substituting the Syscall Queue with a network adapter that sends the data to the server. Moreover, this endeavour does not require NFA states to be approved before being allowed to continue, hence it is possible to authorise system calls as soon as all their data has been collected, greatly improving the application execution speed.

This thesis does not implement the full architecture, which is left for future developments but wants to assess the feasibility of this approach and how effective it is. Therefore, two main tests will be executed, the first will aims to assess if attaching a Java debugger to an Android application would generate execution anomalies, to do this, the NFA model previously learnt in Section 5.2.2 will be reused; a second test will measure the time between one system call and the next one, to validate that there is effectively a significant and noticeable difference.

5.3.2 Debugger as an anomaly

This test has been executed by leveraging Android Studio, which is the most common IDE for Android application development, to automatically run the application in debug mode and attach to it using the Java debugger. Android Studio also offers the possibility to debug C and C++ code, but in this case, this would not be possible since all the used debuggers leverage on `ptrace`, but the role of tracer is already taken by `Ptracer`.

Once the application was started from Android Studio, *Ptracer* was attached to it using the same command used to enforce the learnt model in Section 5.2.3.
The result was that starting from the first system call, which is always an `epoll_pwait` awaiting new events on the dedicated queue, an anomaly was found.
It has been reported in Figure 5.13, where it is possible to see that the stack trace has never been observed before. The reason behind it is that the library used to unwind the stack has not managed to extract a function name for all the frames, and it has stopped halfway, as can be seen in line 17, where it has not managed to extract the function name and exited.
It can be hypothesised that the stack frame format is altered when using a debugger, even if no breakpoints or custom variable evaluations have been used.

Another performed test consists in attaching a debugger to a running instance of *AudioRecorder*.
This was done by executing the following operations in order:

1. Start the application via the Activity Manage command like utility `am`, as described in Section 5.2.1;

2. Attaching `Ptracer` and ensuring that no anomalies are detected after creating a recording and replaying it;

```
----------------- SYSCALL ENTRY START -----------------
Notification origin: it.matteodegiorgi.audiorecorder
PID: 13108
SPID: 13108
Timestamp: 1676798934727175
NOT Authorized
Syscall = epoll_pwait (22)
Stack unwinding = {
  PC 0x00000075f00ce498 Relative PC 0x000000000009e498 SP 000000007fe4bc1880 - __epoll_pwait @ 8
  PC 0x00000075ed52f628 Relative PC 0x0000000000016628 SP 000000007fe4bc1880 - android::Looper::pollInner(int
       ) @ 180
  PC 0x00000075ed52f50c Relative PC 0x000000000001650c SP 000000007fe4bc1a30 - android::Looper::pollOnce(int,
        int*, int*, void**) @ 112
  PC 0x00000075ed77ebb0 Relative PC 0x0000000000157bb0 SP 000000007fe4bc1a70 - android::
       android_os_MessageQueue_nativePollOnce(_JNIEnv*, _jobject*, long, int) @ 44
  ...
  PC 0x000000733e1d574c Relative PC 0x00000000003d574c SP 000000007fe4bc2cc0 - art::interpreter::Execute(art
       ::Thread*, art::CodeItemDataAccessor const&, art::ShadowFrame&, art::JValue, bool, bool) @ 304
  PC 0x000000733e537714 Relative PC 0x0000000000737714 SP 000000007fe4bc2d80 - artQuickToInterpreterBridge @
       776
  PC 0x000000733e022488 Relative PC 0x0000000000222488 SP 000000007fe4bc2f60 -
       art_quick_to_interpreter_bridge @ 88
  PC 0x000000733e02269c Relative PC 0x000000000022269c SP 000000007fe4bc3040
}
Parameters = {
  0x000000000000003e
  0x0000007fe4bc18c0
  0x0000000000000010
  0x00000000ffffffff
  0000000000000000
  0x0000000000000008
  0x00000073df072d70
  0000000000000000
}
Registers = {
  PC: 0x00000075f00ce498
  SP: 0x0000007fe4bc1880
  RET: 0x000000000000003e
}
----------------- SYSCALL ENTRY STOP -----------------
State not found in the list of associations -> Not authorised
Warning! Encountered a state that has never been observed before!

Possible actions:
1 - Kill the target process
2 - Allow it by adding a new state in the model
Choice:
```

Figure 5.13: Anomaly detected in the application started by the Java debugger

3. While the running application was waiting for new events, the Java Debugger was attached by selecting *AudioRecorder* in Android Studio.

From *Ptracer*, it was observed that attaching the Java debugger did not trigger any system call in the application. The reason behind this is that not all of its processes are traced since the same attach procedure described in Section 5.2.1 was followed.
In case all the application child processes are traced, then it would be possible to see a new connection to the JDWP (Java Debug Wire Protocol) agent, which is the process responsible for executing debugger commands.
Now that both *Ptracer* and the Java debugger are attached, it is possible to trigger a first event, which was done by pressing the start recording button. This operation immediately halted the application since an anomaly was found at the first executed system call, which is reported in Figure 5.14.
The reason behind this is the same as before: the stack trace is altered and truncated after attaching the Java debugger, as can be seen in line 18.

```
----------------- SYSCALL ENTRY START -----------------
Notification origin: it.matteodegiorgi.audiorecorder
PID: 13109
SPID: 13109
Timestamp: 1676799535175863
NOT Authorized
Syscall = recvfrom (207)
Stack unwinding = {
  PC 0x00000075f00ce058 Relative PC 0x000000000009e058 SP 000000007fe4bc1d80 - recvfrom @ 8
  PC 0x00000075e72900b4 Relative PC 0x00000000000440b4 SP 000000007fe4bc1d80 - android::InputChannel::
        receiveMessage(android::InputMessage*) @ 48
  PC 0x00000075e72915f4 Relative PC 0x00000000000455f4 SP 000000007fe4bc1db0 - android::InputConsumer::
        consume(android::InputEventFactoryInterface*, bool, long, unsigned int*, android::InputEvent**) @ 196
  PC 0x00000075ed74502c Relative PC 0x000000000011e02c SP 000000007fe4bc1ed0 - android::
        NativeInputEventReceiver::consumeEvents(_JNIEnv*, bool, long, bool*) @ 288
  PC 0x00000075ed744e38 Relative PC 0x000000000011de38 SP 000000007fe4bc1fa0 - android::
        NativeInputEventReceiver::handleEvent(int, int, void*) @ 180
  PC 0x00000075ed52f904 Relative PC 0x0000000000016904 SP 000000007fe4bc1ff0 - android::Looper::pollInner(int
        ) @ 912
  PC 0x00000075ed52f50c Relative PC 0x000000000001650c SP 000000007fe4bc21a0 - android::Looper::pollOnce(int,
         int*, int*, void**) @ 112
  PC 0x00000075ed77ebb0 Relative PC 0x0000000000157bb0 SP 000000007fe4bc21e0 - android::
        android_os_MessageQueue_nativePollOnce(_JNIEnv*, _jobject*, long, int) @ 44
  PC 0x0000000070f42fb8 Relative PC 0x00000000001a5fb8 SP 000000007fe4bc2200 - art_jni_trampoline @ 120
  PC 0x000000733e02269c Relative PC 0x000000000022269c SP 000000007fe4bc22b0
}
Parameters = {
  0x000000000000004e
  0x000000748f074368
  0x0000000000000920
  0x0000000000000040
  000000000000000000
  000000000000000000
  0xff2f1f3c3b1f6479
  000000000000000000
}
Registers = {
  PC: 0x00000075f00ce058
  SP: 0x0000007fe4bc1d80
  RET: 0x000000000000004e
}
----------------- SYSCALL ENTRY STOP -----------------
State not found in the list of associations -> Not authorised
Warning! Encountered a state that has never been observed before!

Possible actions:
1 - Kill the target process
2 - Allow it by adding a new state in the model
Choice:
```

Figure 5.14: Anomaly detected in the application after the Java debugger attached

The final result is satisfactory since it shows that a debugger can easily be spotted by the altered stack traces, but a deeper investigation is needed to understand the exact reason why `libunwindstack` is not able to fully unwind the stack when a Java debugger is attached.
This demonstrates that *Ptracer* could be used to detect debuggers, considering them anomalies from the normal program execution. Nevertheless, the tracer used in isolation does not guarantee to detect and counter debuggers since it would be vulnerable to a MATE attacker, who could terminate it, but it is necessary to leverage an architecture that makes the tracing data a requisite for the application usage. For example, the proposed architecture, where the tracing data is constantly evaluated by a server to decide whether the client is being tampered with.

5.3.3 Time checks

This section aims to test whether the Java debugger, when attached, produces a significant slowdown of the traced application. To do that, the time between system calls will be measured in microseconds (10^{-6} of a second) and compared between multiple runs.
More specifically, the *AudioRecorder* application will be executed three times to measure times in the following different conditions:

1. Only with *Ptracer* attached and no Java debugger.
2. With the Java debugger attached but without any breakpoint set.
3. With the Java debugger attached and one breakpoint set in the start recording button listener.

To perform the first test, the application will be started using the `am` command line utility, while for the other two, Android Studio will be used to start the application in debug mode and automatically attach to it.

To collect the execution times between system calls, *Ptracer* has been slightly modified by adding an additional Hash Map that maps TIDs and exit timestamps, and for each system call notification received through the Syscall Queue:

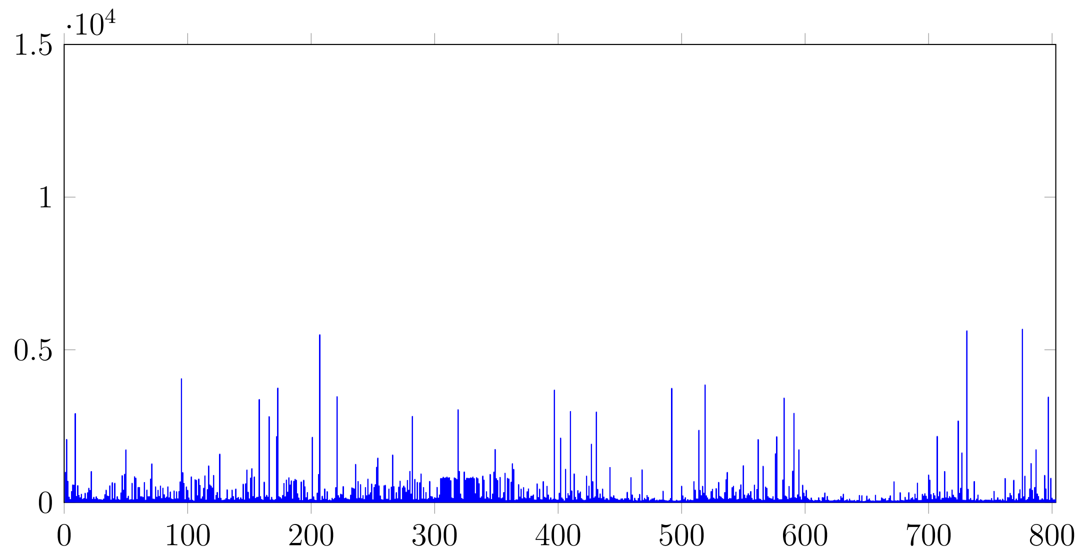
- If it is an entry notification, then look for its TID in the map, if present, then subtract the associated timestamp from the timestamp of the entry notification and print it on standard output.
- If it is an exit notification, then save its TID and timestamp in the Hash Map.

In this way, the time between a system call and the subsequent one is measured, i.e. the time between a system call exit and the notification of the next entry for the same TID. It would not make sense to measure the time between a system call entry and exit since that time is used by the kernel and is also influenced by the user's behaviour.
After disabling the usual printing of the received notifications, it is now possible to execute the three tests and plot the obtained data and start analysing them.

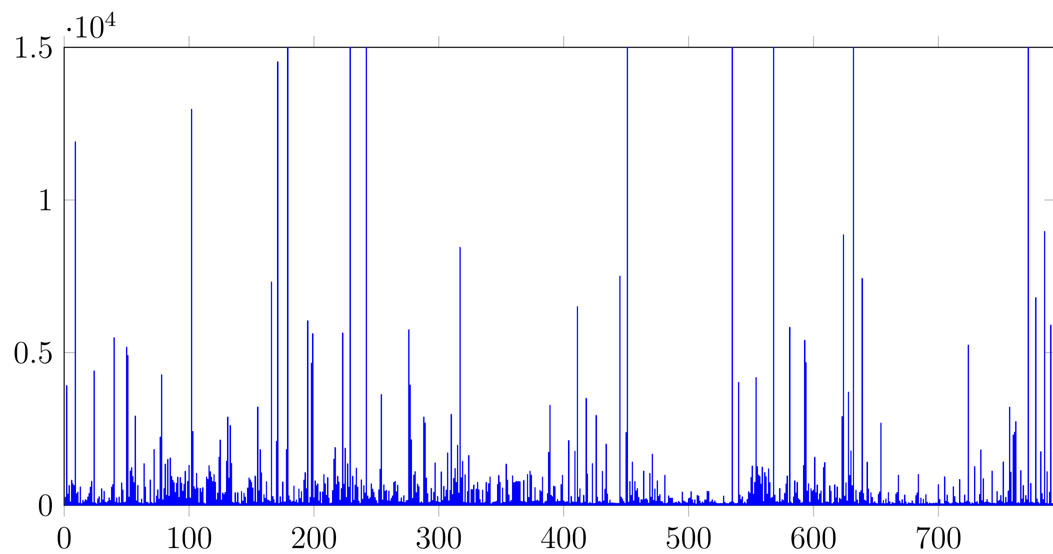
It has been decided to perform the same user actions for each test, hence pressing all four buttons: starting a recording, pausing it and then replaying the audio file and stop replaying.

The three test results have been plotted and reported below, where it is clearly possible to see that Figure 5.15a, where the first test results are reported, shows faster execution times than Figure 5.15b and Figure 5.15c, where a debugger was used.
It can also be noticed that setting a breakpoint does not have a noticeable impact.

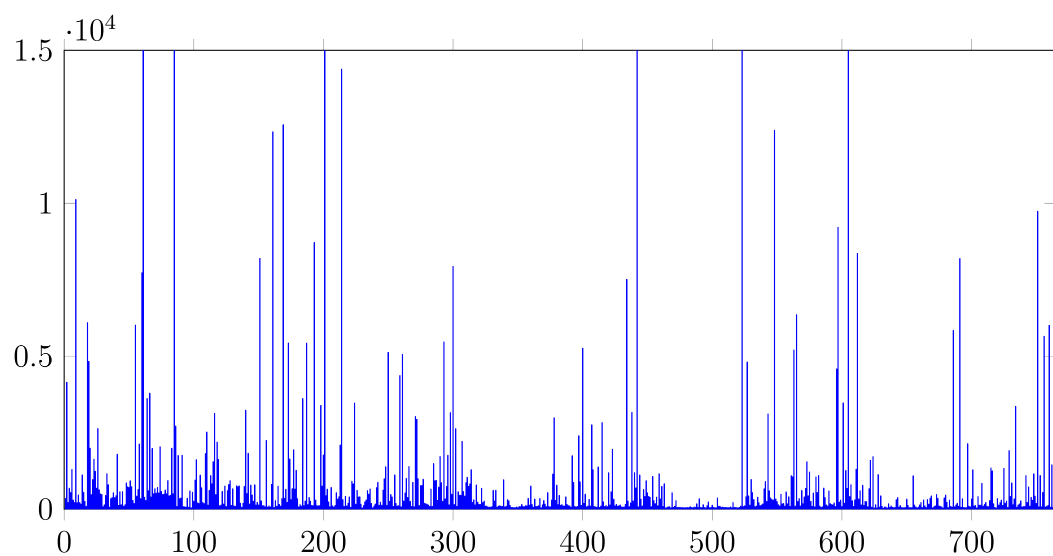
More data on the collected datasets can be seen in Table 5.2, where, for each test, some statistics on the collected timings between the observed system calls.
Also from this data, it is evident how the first test, without a debugger, is considerably faster than the other two. Moreover, it is also possible to see that the average time of the third test, where an extra breakpoint has been set, is slightly higher than the second.



(a) Plot showing time elapsed between system calls with no Java debugger attached



(b) Plot showing time elapsed between system calls when a Java debugger is attached



(c) Plot showing time elapsed between system calls when a Java debugger is attached and a breakpoint set

Figure 5.15: Plots showing the different execution times when a debugger is attached

Test	Average	Median	Standard Deviation	Range	Samples
1	399,918	163	644,683	28 - 5667	803
2	921,925	267.5	2598,898	28 - 39832	794
3	962,643	251	2939,837	27 - 39775	765

Table 5.2: Statistics on the timings collected during the three tests

The final result of these tests clearly highlights that a Java debugger, when attached to a process, produces a considerable slowdown, which could be identified as an anomaly using statistical methods. Future developments in this regard will be proposed in Section 6.6.

5.4 Privacy issues

This section aims to validate that *Ptracer* can be used to identify privacy issues while tracing an Android application.
It is considered a privacy issue, every operation that acquires or exfiltrates user data in a covert fashion and unknowingly from the user.

To validate this capability, *Ptracer*'s System Call Decoders will be heavily used since they provide a deeper insight into the application's behaviour. They have been described in Section 4.5, where more technical details about their implementation can be found.

To validate this feature, the System Call Decoders will be activated tracing *AudioRecorder* to validate that the following information can be seen:

1. The microphone is accessed via the Android audio service;
2. The recording file is written on the external memory;
3. There are no external internet connections;
4. No external processes have been executed.

The first two of these actions are executed by interacting with the Android Binder IPC, which allows to delegate them to other Android services. In fact, Android does not allow direct access to resources like the microphone, but it requires every application to go through the available services and their interface. Therefore, most of the above actions will boil down to IPC method calls.

To run and attach to *AudioRecorder*, the same method described in Section 5.2.1 will be used, therefore, the same considerations about not being able to capture all the system calls apply also here.
It is now possible to disable many *Ptracer* options that were used before since they are not needed anymore, this will result in a speedup of the application execution. To do that, the following command is used:

```
./ptracer --decoders true \     # Enables the system call decoders
          --authorizer false \  # Disable the Authorizer component
          --backtrace false \   # Not necessary when not building the NFA model
          --pid $(ps -A | grep matteo | awk '{ print $2;}') # Find the PID to trace
```

Where the decoders are explicitly enabled, the Authorizer is disabled since it is desired to authorise all the system calls immediately, and the stack unwinder is also disabled since stack information is not needed in this context.

During the first application execution, it was possible to identify several calls to the Android service `media.player`, which is implemented by the interface named `android.media.IMediaRecorder`. *AudioRecorder* obtains a handle to communicate with the media recorder, whose value is 50 and initiates to configure it to start recording.

```
Sent:
  Protocol: 0x40406300 (BC_TRANSACTION)
    Target: 0x32 (50)
    Code: 16
    Flags: 16 (TF_ACCEPT_FDS)
    Buffer pointer: 0x73af054ac0 (496857598656), Data size: 100
    Buffer content:
    0x73af054ac0: 04 00 00 c2 ff ff ff ff 54 53 59 53 1c 00 00 00 ........ TSYS....
    0x73af054ad0: 61 00 6e 00 64 00 72 00 6f 00 69 00 64 00 2e 00 a.n.d.r. o.i.d...
    0x73af054ae0: 6d 00 65 00 64 00 69 00 61 00 2e 00 49 00 4d 00 m.e.d.i. a...I.M.
    0x73af054af0: 65 00 64 00 69 00 61 00 52 00 65 00 63 00 6f 00 e.d.i.a. R.e.c.o.
    0x73af054b00: 72 00 64 00 65 00 72 00 00 00 00 00 85 2a 64 66 r.d.e.r. .....*df
    0x73af054b10: 7f 01 00 00 60 00 00 00 00 00 00 00 00 00 00 00 ....`... ........
    0x73af054b20: 00 00 00 00                                     ....
    Interface: android.media.IMediaRecorder
    Offsets pointer: 0x735f07ba00 (495515580928), Offsets size: 8
    Offsets content:
    0x735f07ba00: 4c 00 00 00 00 00 00 00                         L.......
    Offset 0:
      Type: 0x66642a85 (BINDER_TYPE_FD)
      File Descriptor: 96

Received:
  Protocol: 0x720c (BR_NOOP)
  Protocol: 0x7206 (BR_TRANSACTION_COMPLETE)
  Protocol: 0x80407203 (BR_REPLY)
    Target: 0x0 (0)
    Sender EUID: 1013
    Buffer pointer: 0x73286fc000 (494599651328), Data size: 4
    Buffer content:
    0x73286fc000: 00 00 00 00                                     ....
    Offsets pointer: 0x73286fc008 (494599651336), Offsets size: 0
------------------ BINDER CALL STOP -------------------
```

Figure 5.16: Binder Decoder snippet showing the request to record on a specific file descriptor

In Figure 5.16, it is possible to see the invocation of method number 16 (visible in line 4) on the media recorder handle (specified as a target in line 3), which sends the file descriptor number 96 (in line 21) to the media player service.

As the File Decoder shows in Figure 5.17 in line 7, this file descriptor corresponds to `AudioRecording.3gp` in the external storage, and after this method invocation, the media player will be able to write directly into it.
After having communicated the file descriptor, *AudioRecorder* will invoke the method `void prepare()`, which has the number 9 and then the method void start(), which has the number 8. These calls have not been reported since they do not carry any extra parameters or return any.

From the final File Decoder report, in Figure 5.17, it is also possible to see that the configuration file was read when the start recording button was pressed.
As expected, the Socket and Execve decoders produced an empty report since none of those system calls are invoked.

```
----------------- FILE DECODER START -----------------
...
File Descriptor: 96 <---> /storage/emulated/0/Android/data/it.matteodegiorgi.audiorecorder/files/config/
    config.bin
Read content extracted in: "./FileDecoder/5942/96-read-1676819124624", bytes: 125
File Descriptor: 71 <---> fd-71
Write content extracted in: "./FileDecoder/5942/71-write-1676819124677", bytes: 16
File Descriptor: 96 <---> /storage/emulated/0/Android/data/it.matteodegiorgi.audiorecorder/files/audio/
    AudioRecording.3gp
----------------- FILE DECODER STOP -----------------
```

Figure 5.17: Snippet of the File Decoder final report after the application has started recording

This result was expected since the tested application is very simple, to be able to test all the decoders on a more complex application, it was decided to use *Ptracer* to monitor a more mature and complex application, like *Instagram*.
Instagram contains a multitude of features which make an advanced usage of many system services, but it also represents one of those applications whose behaviour is uncertain from a privacy point of view. To monitor it, the same approach as before has been used, but this time the phone internet connection was disabled until *Ptracer* attached to the application.

After just 3 minutes of usage, the application has generated more that 90 thousand system calls and 1300 interactions with the Android Binder. Given the large amount of these calls, it was not possible to provide the same level of detail as with the previous application, but *Ptracer* can automatically extract the name of the interfaces that the application communicated with, they can be seen in Figure 5.18.

```
android.app.IActivityManager
android.app.INotificationManager
android.app.job.IJobCallback
android.app.job.IJobScheduler
android.content.IContentProvider
android.content.pm.IPackageManager
android.gui.DisplayEventConnection
android.gui.ITransactionComposerListener
android.location.ILocationManager
android.net.IConnectivityManager
android.os.IMessenger
android.os.IPowerManager
android.os.IServiceManager
android.os.storage.IStorageManager
android.ui.ISurfaceComposer
android.view.IWindowSession
com.android.internal.view.IInputMethodManager
com.facebook.push.fbns.ipc.IFbnsAIDLService
```

Figure 5.18: List of interfaces used by *Instagram* during the test

From the list of interfaces, it is possible to see some significant names, like the location service `android.location.ILocationManager`, which can be used to retrieve the last geographical location of the phone.
Digging deeper into what methods have been called on this interface, it was possible to find the transaction reported in Figure 5.19. This transaction calls method number 1 in the AIDL interface `ILocationManager`, as can be seen in line 5, and the service replies with an object, which at the moment is not extracted by *Ptracer*. The invoked method signature can be seen in the code snippet reported in Fig-

ure 5.20, and it is clear that it can be used to acquire the last geographical location of the phone.

```
1   ------------------ BINDER CALL START ------------------
2   Sent:
3    Protocol: 0x40406300 (BC_TRANSACTION)
4     Target: 0x43 (67)
5     Code: 1 (FLAG_ONEWAY)
6     Flags: 16 (TF_ACCEPT_FDS)
7     Buffer pointer: 0x73ef136890 (497932265616), Data size: 164
8     Buffer content:
9     0x73ef136890: 04 00 00 c2 ff ff ff ff 54 53 59 53 21 00 00 00 ........ TSYS!...
10    0x73ef1368a0: 61 00 6e 00 64 00 72 00 6f 00 69 00 64 00 2e 00 a.n.d.r. o.i.d...
11    0x73ef1368b0: 6c 00 6f 00 63 00 61 00 74 00 69 00 6f 00 6e 00 l.o.c.a. t.i.o.n.
12    0x73ef1368c0: 2e 00 49 00 4c 00 6f 00 63 00 61 00 74 00 69 00 ..I.L.o. c.a.t.i.
13    0x73ef1368d0: 6f 00 6e 00 4d 00 61 00 6e 00 61 00 67 00 65 00 o.n.M.a. n.a.g.e.
14    0x73ef1368e0: 72 00 00 00 03 00 00 00 67 00 70 00 73 00 00 00 r....... g.p.s...
15    0x73ef1368f0: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........ ........
16    0x73ef136900: 15 00 00 00 63 00 6f 00 6d 00 2e 00 69 00 6e 00 ....c.o. m...i.n.
17    0x73ef136910: 73 00 74 00 61 00 67 00 72 00 61 00 6d 00 2e 00 s.t.a.g. r.a.m...
18    0x73ef136920: 61 00 6e 00 64 00 72 00 6f 00 69 00 64 00 00 00 a.n.d.r. o.i.d...
19    0x73ef136930: ff ff ff ff                                     ....
20    Interface: android.location.ILocationManager
21
22  Received:
23   Protocol: 0x720c (BR_NOOP)
24   Protocol: 0x7206 (BR_TRANSACTION_COMPLETE)
25   Protocol: 0x80407203 (BR_REPLY)
26    Target: 0x0 (0)
27    Sender EUID: 1000
28    Buffer pointer: 0x72ca4df000 (493020377088), Data size: 8
29    Buffer content:
30    0x72ca4df000: 00 00 00 00 00 00 00 00                         ........
31    Offsets pointer: 0x72ca4df008 (493020377096), Offsets size: 0
32  ------------------ BINDER CALL STOP -------------------
```

Figure 5.19: RPC performed by *Instagram* on the Android location service

```
1   package android.location;
2
3   ...
4
5   interface ILocationManager
6   {
7       @nullable Location getLastLocation(String provider,
8                                          in LastLocationRequest request,
9                                          String packageName,
10                                         @nullable String attributionTag);
11      ...
12  }
```

Figure 5.20: Code snippet from the AIDL interface `ILocationManager`

Another potentially interesting service the application interacts with is the following: `com.facebook.push.fbns.ipc.IFbnsAIDLService`. In this case, there is only one transaction directed to this service, and it has been reported in Figure 5.21. Since the AIDL interface exposed by *Facebook* is not public, then it is not possible to know more information about the called method. But, given the extracted information, it can be assumed that it is related to the communication of a "Analytics UID", which can be hypothesised related to advertisements.

Although it would have been possible to go deeper into the actions performed by the traced applications, it has been decided to stop since the initial objectives have

```
1  ------------------ BINDER CALL START ------------------
2  Sent:
3    Protocol: 0x40406300 (BC_TRANSACTION)
4      Target: 0x56 (86)
5      Code: 2
6      Flags: 17 (TF_ONE_WAY | TF_ACCEPT_FDS)
7      Buffer pointer: 0x742f04ed50 (499005058384), Data size: 484
8      Buffer content:
9      0x742f04ed50: 00 00 00 80 ff ff ff ff 54 53 59 53 2b 00 00 00 ........ TSYS+...
10     0x742f04ed60: 63 00 6f 00 6d 00 2e 00 66 00 61 00 63 00 65 00 c.o.m... f.a.c.e.
11     0x742f04ed70: 62 00 6f 00 6f 00 6b 00 2e 00 70 00 75 00 73 00 b.o.o.k. ..p.u.s.
12     0x742f04ed80: 68 00 2e 00 66 00 62 00 6e 00 73 00 2e 00 69 00 h...f.b. n.s...i.
13     0x742f04ed90: 70 00 63 00 2e 00 49 00 46 00 62 00 6e 00 73 00 p.c...I. F.b.n.s.
14     0x742f04eda0: 41 00 49 00 44 00 4c 00 53 00 65 00 72 00 76 00 A.I.D.L. S.e.r.v.
15     0x742f04edb0: 69 00 63 00 65 00 00 00 01 00 00 00 fa 84 d1 01 i.c.e... ........
16     0x742f04edc0: 00 00 00 00 64 01 00 00 42 4e 44 4c 05 00 00 00 ....d... BNDL....
17     0x742f04edd0: 20 00 00 00 4c 00 4f 00 47 00 47 00 49 00 4e 00 ...L.O. G.G.I.N.
18     0x742f04ede0: 47 00 5f 00 48 00 45 00 41 00 4c 00 54 00 48 00 G._.H.E. A.L.T.H.
19     0x742f04edf0: 5f 00 53 00 54 00 41 00 54 00 53 00 5f 00 53 00 _.S.T.A. T.S._.S.
20     0x742f04ee00: 41 00 4d 00 50 00 4c 00 45 00 5f 00 52 00 41 00 A.M.P.L. E._.R.A.
21     0x742f04ee10: 54 00 45 00 00 00 00 00 01 00 00 00 1e 00 00 00 T.E..... ........
22     0x742f04ee20: 0f 00 00 00 41 00 4e 00 41 00 4c 00 59 00 54 00 ....A.N. A.L.Y.T.
23     0x742f04ee30: 49 00 43 00 5f 00 46 00 42 00 5f 00 55 00 49 00 I.C._.F. B._.U.I.
24     0x742f04ee40: 44 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 D....... ........
25     0x742f04ee50: 13 00 00 00 41 00 4e 00 41 00 4c 00 59 00 54 00 ....A.N. A.L.Y.T.
26     0x742f04ee60: 49 00 43 00 5f 00 59 00 45 00 41 00 52 00 5f 00 I.C._.Y. E.A.R._.
27     0x742f04ee70: 43 00 4c 00 41 00 53 00 53 00 00 00 01 00 00 00 C.L.A.S. S.......
28     0x742f04ee80: ff ff ff ff 14 00 00 00 41 00 4e 00 41 00 4c 00 ........ A.N.A.L.
29     0x742f04ee90: 59 00 54 00 49 00 43 00 5f 00 49 00 53 00 5f 00 Y.T.I.C. _.I.S._.
30     0x742f04eea0: 45 00 4d 00 50 00 4c 00 4f 00 59 00 45 00 45 00 E.M.P.L. O.Y.E.E.
31     0x742f04eeb0: 00 00 00 00 09 00 00 00 00 00 00 00 0c 00 00 00 ........ ........
32     0x742f04eec0: 41 00 4e 00 41 00 4c 00 59 00 54 00 49 00 43 00 A.N.A.L. Y.T.I.C.
33     0x742f04eed0: 5f 00 55 00 49 00 44 00 00 00 00 00 00 00 00 00 _.U.I.D. ........
34     0x742f04eee0: 24 00 00 00 38 00 39 00 61 00 39 00 32 00 32 00 $...8.9. a.9.2.2.
35 ---------- REDACTED ANALYTICS UID ----------
36     0x742f04ef20: 66 00 34 00 35 00 63 00 34 00 30 00 00 00 00 00 f.4.5.c. 4.0.....
37     0x742f04ef30: 04 00 00 00                                     ....
38     Interface: com.facebook.push.fbns.ipc.IFbnsAIDLService
39
40 ...
41 ------------------ BINDER CALL STOP -------------------
```

Figure 5.21: RPC performed by *Instagram* on a *Facebook* AIDL service

been achieved.
Ptracer can provide useful information to detect privacy issues, but it requires to be improved implementing automatic ways to correlate Binder transactions and increasing the set of analysed system calls. More details will be discussed in the future developments section 6.2.

Chapter 6

Future Developments

This chapter will discuss what the proposed future developments are, effectively formalizing what are the lessons learnt during the development and validation of *Ptracer*.

6.1 eBPF

The Extended Berkeley Package Filter (eBPF) is a Linux kernel technology (available since 4.4) which allows running sandboxed programs in the kernel without altering the kernel source code.
An eBPF program can register itself to various kernel hooks, allowing it to retrieve information about system calls, function entries and exits, network events and kprobes or uprobes (probes for kernel or user functions). For each of these data categories, an eBPF can collect a multitude of data points for the whole system and have the chance to make decisions such as dropping a network packet or modifying a system call parameter.
Moreover, they can send the collected data to another program in the userland, which could perform more advanced operations since the eBPF program has a limited size and has to formally guarantee its termination before being loaded by the kernel.

One possible future development of *Ptracer* is supporting inputs from an eBPF probe custom-made for the purpose. This approach would guarantee an increased speed but would not allow holding system calls one by one until the model decides whether they are authorised or not.
Despite this would imply a lowered security level, it can be a good solution for adapting the tracer program to a world made of more and more complex programs which cannot accept to be slowed down more than a certain threshold.

6.2 System Call Decoders

The implemented decoders should be intended as a proof of concept of the level of insight into an application behaviour reachable by analysing its system calls parameters.
For this reason, not all the system calls have been treated, and some significant ones are missing. For instance, the system calls `name_to_handle_at` and

`open_by_handle_at` are not handled by the File Decoder yet and could allow opening files. Another system call not fully implemented is `execveat`, where the parameters `dirfd` and `flags` are not taken into consideration by the relative decoder and can significantly change its behaviour.
Other examples of missing decoding are available (e.g., `mmap` should also be analysed), and more should be discovered, hence a thorough analysis is needed.

One key point in making *Ptracer* able to analyse commercial applications is improving the Binder IPC Decoder. At the current status, its implementation does not deeply understand the exchanged data and not all the Parcel types are understood.
Since the Binder is such a central point in every Android system, one important future development is not only being able to parse complete transactions and give them their correct meaning (e.g., identify also the called interface method) but also correlate the various Remote Procedure Calls or exchanged Intents. This could lead to the generation of a graph that maps all the communications performed by an application.

Eventually, all the information collected from the decoders could be captured in the NFA model to provide an even more accurate description of the program behaviour.

6.3 Attach to Zygote

One of the current main limitations of *Ptracer* is its inability to trace applications from their initialization, hence missing potentially important system calls. A way to solve this issue is to attach to the child of the Zygote process that will specialise in the desired application.

When the Zygote process receives a request to launch a new application, it will select one of its unspecialized forks from the Blastula pool and communicate to it the package and activity name it shall start.
Both these communications take place on UNIX sockets, thus, it would be possible to trace the Zygote process and intercept the communication between it and its fork, where all the application details are sent. Hence, when the desired application is started, *Ptracer* could trace the target application from its initialization phase.

6.4 Improve the tracing mechanism

At the moment, when starting to trace an application, *Ptracer* will attach only the specified Thread ID. When tracing more complex applications, it would be necessary to automatically attach to the entirety of their process tree, or at least leave the final choice to the user.

Moreover, one of the current limitations is the slowdown imposed by the stack analysis since it is a very demanding operation and requires multiple reads from the tracee's memory. Especially in Android, where stack traces can easily reach hundreds of entries.
For these reasons, it shall be considered to allow heuristics that allow stopping parsing the stack frames earlier, for example, it may not always be needed to

reach the function `main` on top of the stack.

While testing *Ptracer*'s debugger detection capabilities in Section 5.3.2 it was discovered that `libunwindstack` is not able to parse certain stack frames when a debugger is attached. This should be investigated further to assess the library's limitations and decide if it is a feature worth implementing.

6.5 Model improvements

While assessing `Ptracer`'s ability to detect anomalies by enforcing its NFA model in Section 5.2.3, many false positives were encountered when *AudioRecorder*'s main activity was recreated. After a quick investigation, it was found that there are many mutexed and synchronization operations happening, thus, depending on the order of events, they would hold the process or not. Which caused a multitude of different system call combinations.
One way to solve this issue, especially while tracing big programs written in interpreted languages, would be to exclude from the NFA all the code that is not directly or indirectly invoked from one of the application's functions.
Alternatively, moving away from the current black-box approach would allow the developers to define only some security-critical functions that should be traced. This could be done via code annotations and deeper integration into the application's logic.

In some cases, the NFA model is too strict for the target application, for example, because of a large number of threads performing synchronisation operations among them and invoking many system calls. In such cases, it might be desired to adopt a more relaxed strategy based on policies instead of an automaton, where generic rules like "This application only writes files in this directory" can be learnt or specified by the user.

The current model is not able to counter exploits that target logic vulnerabilities or denial of service attacks.
A future development could be introducing into the model also information about the system call parameters, which could be acquired from the dedicated decoders. This would allow to capture information like: "The executable path passed to an `execve` is always x ".
To prevent denial of service attacks, it would be possible to include statistical information in the model, effectively transforming it into a Markov chain.

6.6 Debugger detection evolutions

Section 5.3 validates that *Ptracer* is suitable to be used to detect debuggers and indirectly prevent them from attaching to an application.
One of the requisites for *Ptracer* to be effective in protecting the application from this kind of MATE attack is that it is used in an architecture that establishes a dependency between the client and server.
A future development is detaching the tracing and analyser components of *Ptracer* in order to place the first inside the client application and the second in the server. In this way, only the server knows the NFA model and will be able to make decisions accordingly.

Moreover, to implement this architecture, it would also be necessary to create a statistical model to identify whether the client is too slow and raise this as an anomaly. This operation needs to happen on the server and could benefit from the previously mentioned effort of linking through the network the two internal components of the tracer or simply be based on the logs produced by it.

Chapter 7

Conclusions

The proposed tool *Ptracer* proved to be a valuable asset for monitoring and analyzing the behaviour of Android applications. By intercepting and recording system calls, it is possible to gain a detailed understanding of the actions the application is taking, identify unusual patterns of behaviour, and detect the presence of debuggers.

The anomaly detection capabilities provided by *Ptracer* are based on a Non-Deterministic Finite Automaton where not only the system call number is captured but also the full stack trace that leads to them.
This model has proven to be effective in protecting Android applications against vulnerabilities and debuggers, but at the same time quite strict when dealing with interpreted languages. This implied that several learning iterations were needed before it stabilized, and some false positives were found when enforced.
Nevertheless, it proved suitable to be enforced over normal application usage and very effective in detecting any diversion from normal behaviour.
Future developments were proposed to make the model suitable also to very complex applications and to be able to identify logic vulnerabilities and denial of service attacks

It was proposed an architecture for detecting debuggers in the scenario of a server and client application, where the server needs to ensure that all its clients are not being tampered with. A client application embeds *Ptracer*, whose role is to trace the application itself and send the resulting data to the server. The latter will be able to decide whether the client is genuine or not by leveraging on the time distance between two system calls and if the NFA model detects any anomaly.
It was successfully validated that in case a Java debugger is attached, then the NFA model would immediately spot an anomaly, given the immediate difference in the generated stack traces. Moreover, when a debugger is attached, it was shown that the execution times between two system calls are significantly slower, easily allowing the detection of an anomaly.
Ptracer is effective in the proposed architecture only when integrated with other multi-layered protections against MATE attacks, e.g., introducing remote attestation from the server.

Ptracer was also evaluated for its capability of identifying privacy issues by deeply analysing system calls extracting and understanding their parameters. This was validated over *AudioRecorder*, a custom Android application developed for the

purpose, and tested over the *Instagram*.
It was possible to validate the effectiveness of this feature over *AudioRecorder* with good results since it was able to highlight the access to the microphone successfully. While over *Instagram*, it was able to identify the fact that it accessed the current phone location and its communication with the *Facebook* application.
Eventually, future developments were proposed to build an even more insightful data model and make it easier to spot privacy issues.

Appendices

Appendix A

Build and installation

This appendix describes how to compile and install *Ptracer*, which supports x86_64 and AArch64 (ARMv8-A) Linux platforms and leverages on `ptrace` to capture System Calls and signals.
Depending on the target platform, it is possible to choose between a version that targets generic Linux distributions and one specific for Android. The main difference between the two is the library used to unwind the stack, since the second contains a library specialised for Android (`libunwindstack`), while the first contains the generic library `libunwind`. Moreover, the Android version includes the Binder IPC decoder.

A.1 Usage

It is mandatory to specify either a command whose execution will be traced or a process to attach to. To specify one or another, it is possible to use the command line options `--run` or `--pid` as follows:

```
$ ./ptracer --run ls -la
```

For example, in case it is desired to trace the *Facebook* application, then it is possible to get its PID using the following command:

```
ps -A | grep facebook
```

Which would produce an output similar to the following:

```
OnePlus6T:/data/local/tmp # ps -A | grep facebook
u0_a373   4580 858 17095988 283656 SyS_epoll_wait 0 S com.facebook.katana
u0_a393   6894 858 16899220 218856 SyS_epoll_wait 0 S com.facebook.orca
```

And then run *Ptracer* as follows:

```
OnePlus6T:/data/local/tmp # ./ptracer --pid 6894
```

Please note that to be able to attach to a process in Android, it is necessary to run as root. Therefore, an Android phone which has been previously "rooted" is needed, or, alternatively, it is possible to use an emulator.

More options are available as described in the help section of the command line interface, whose output can be invoked by using the `--help` parameter and is reported below:

```
$ ./ptracer --help
Ptracer usage:
  --help              Display this help message
  --pid arg           PID of the process to trace
  --run arg           Run and Trace the specified program with
                      parameters, if specified, needs to be the last
                      option
  --follow-threads arg (=1) Trace also child threads
  --follow-children arg (=1) Trace also child processes
  --jail arg (=0)     Kill the traced process and all its children if
                      ptracer is killed
  --decoders arg (=1) Enables or disables system call decoders
  --backtrace arg (=1) Extract the full stacktrace that lead to a
                      systemcall
  --authorizer arg (=0) Enable or disables the Authorizer module and all
                      its options
  --learn arg (=1)    Sets the Authorizer module in learning mode
  --nfa arg           Specifies the path where the NFA managed by the
                      Authorizer is present or will be created
  --dot arg           Specifies the path where the DOT representation of
                      the NFA managed by the Authorizer will be created
  --associations arg  Specifies the path where the associations between
                      state IDs and System Calls are present or will be
                      created by the Authorizer
  --name arg          Name of the executable to attach to, used only
                      when a PID is specified
```

For example, to attach to a specific SPID (which is a Thread ID in Linux terms) and:

- follows all the threads that it will generate;
- follows all the processes that it will generate;
- terminate it if *Ptracer* is terminated.

Then the following command can be used:

```
./ptracer --follow-threads true --follow-children true --jail true --backtrace true --pid 6894
```

The Authorizer module can be used to generate a model of the observed behaviour of a program in the form of an NFA. This module can be in “learn” or “enforce” mode, in the first case, it will create an NFA based on the observed behaviour, in the second, it will stop the tracee every time the combination of a System Call and its stack trace has never been encountered before, or the transition between two states has not been observed before.
In the generated NFA, every state corresponds with a System Call number together with the Stack Trace that has to lead to its generation (if not explicitly disabled with the `--backtrace` option).

To use the Authorizer module, it is necessary to specify at least the location where the NFA should be saved and the location where the list of associations between NFA states and the combination of $(SystemCall, StackTrace)$ will be saved. Optionally it is possible to generate the DOT representation of the NFA if the `--dot` option has been specified.

The following command can be used to run the command ls -la, learn a new NFA from its execution and save its DOT representation:

```
./ptracer --authorizer true --learn true --nfa ls.nfa --dot ls.dot --associations ls.ass --run ls -la
```

Please note that attaching to a process in the middle of its execution might result in unstable results when using the Authorizer module.

The project also contains some System Call Decoders, which will provide a final report with a deeper analysis of some system calls. They are active by default, if they are not needed, the program execution can be speedup by disabling them using the option `--decoders false`.

The parameter `--name` can be used when it is desired to attach to a running process to specify its name. It is needed every time the Authorizer component is active, since it will need a name to associate to the states the will be saved in the model.

A.2 Dependencies

The project dependencies tree for Acan be seen below:

```
ptracer
  boost 1.80.0 (Conan)
  android-ndk r25 (Conan)
  libunwind 1.6.2 (Conan) (Only in x86_64 on Linux generic)
  libalf
  libunwindstack-ndk
    lzma-ndk
    libdexfile-ndk
      libartbase-ndk
        libziparchive-ndk
          libbase-ndk
        libtinyxml2
        libartpalette-ndk
        libbase-ndk
        liblog-ndk
          libcutils-ndk
          libutils-ndk
          libsystem-ndk
        libcap-official
      libartpalette-ndk
      libziparchive-ndk
        libbase-ndk
      libbase-ndk
```

Some of these dependencies are provided by *Conan*, which is the Package Manager for C++, while all the other dependencies had to be manually integrated into the project.
A big part of this effort was related to extracting all the dependency tree of `libunwindstack` from the Android source code. Multiple adaptations were necessary and all of its dependencies are currently hosted together with *Ptracer*.

A.3 Build

The following tools are expected to be pre-installed to be able to compile the project:

- *Conan* version 1.50 or above: Used to handle dependencies like Boost and Android NDK.
- *CMake* version 3.23 or above: Use to control the software compilation and link together all the other dependencies.

Depending on what architecture you are targeting, it is possible to use the following build scripts in order to invoke Conan and CMake properly:

- `./build/aarch64/build.sh`: Used to build an executable and statically linked library for ARMv8 architectures

- `./build/x86_64/build.sh`: Used to build an executable and statically linked library for x86_64 architectures but not Android
- `./build/x86_64-android/build.sh`: Used to build an executable and statically linked library for x86_64 architectures running Android

Once the build process has terminated, the `ptracer` executable file will be in `./build/$ARCH/cmake-build-debug/bin`, and the statically and dynamically linked libraries will be in `./build/$ARCH/cmake-build-debug/lib`.

It has been necessary to subdivide the build for x86_64 architectures running Android and not because the last ones will benefit from the stack unwinding capabilities of `libunwindstack` and to do that, it requires to be compiled using Android NDK.

A.4 Debug

The project uses the user-defined signal `SIGUSR1`, and by default, GDB will stop at every signal, to modify this behaviour it is possible to use the following GDB command:

```
handle SIGUSR1 nostop noprint pass
```

This can be done automatically by putting this command into the file `/.gdbinit`.

Debugging native Android applications can be done using the GDB server, which can be found in the Android NDK distribution and copied to the Android device via:

```
adb push $ANDROID_SDK/ndk-bundle/prebuilt/android-arm64/gdbserver/gdbserver /data/local
    /tmp/
```

The folder /data/local/tmp/ is used since executables in there are allowed to be executed.

The GDB Server can be used as follows:

```
./gdbserver --once 0.0.0.0:7777 ./ptracer --run ls -la
```

If the Android system is not directly reachable (e.g., it is an emulated instance in Android Studio), it is possible to forward a socket connection using the `adb` utility tool. For example, to forward the local TCP port 5000 to the Android TCP port 5001, it is possible to use the following command:

```
adb forward tcp:5000 tcp:50001
```

For more information regarding `adb` and forwarding, please check the related section in the `adb` manual.

Bibliography

[1] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. Tightly-coupled self-debugging software protection. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450348416. doi: 10.1145/3015135.3015142. URL `https://doi.org/10.1145/3015135.3015142`.

[2] Bert Abrath, Bart Coppens, Jens Van Den Broeck, Brecht Wyseur, Alessandro Cabutto, Paolo Falcarin, and Bjorn De Sutter. Code renewability for native software protection. *ACM Trans. Priv. Secur.*, 23(4), aug 2020. ISSN 2471-2566. doi: 10.1145/3404891. URL `https://doi.org/10.1145/3404891`.

[3] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. pages 15–26, 10 2011. ISBN 9781450310000. doi: 10.1145/2046614.2046619.

[4] Alessandro Cabutto, Paolo Falcarin, Bert Abrath, Bart Coppens, and Bjorn De Sutter. Software protection with code mobility. pages 95–103, 10 2015. doi: 10.1145/2808475.2808481.

[5] Mueller Bernhard Carlos Holguera, Schleier Sven. *OWASP Mobile Application Security Testing Guide*, 1.5.0 edition, 09 2022. URL `https://github.com/OWASP/owasp-mastg`.

[6] Abhishek Chaturvedi, Sandeep Bhatkar, and Ramachandran Sekar. Improving attack detection in host-based ids by learning properties of system call arguments. 01 2005.

[7] Christian Collberg. Tigress homepage. `https://tigress.wtf`, 2020. Accessed: 02-01-2023.

[8] Christian S. Collberg, Jack W. Davidson, Roberto Giacobazzi, Yuan Xiang Gu, Amir Herzberg, and Fei-Yue Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26:8–13, 2011.

[9] ASPIRE Consortium. Aspire project. `https://aspire-fp7.eu`, 2016. Accessed: 02-01-2023.

[10] David Mosberger-Tang Dave Watson, Arun Sharma. Libunwind homepage. `https://www.nongnu.org/libunwind/index.html`, 2020. Accessed : 13-01-2023.

[11] Bjorn De Sutter, Paolo Falcarin, Brecht Wyseur, Cataldo Basile, Mariano Ceccato, Jerome d'Annoville, and Michael Zunke. A reference architecture for software protection. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 291–294, 2016. doi: 10.1109/WICSA.2016.43.

[12] Biniam Fisseha Demissie, Mariano Ceccato, and Roberto Tiella. Assessment of data obfuscation with residue number coding. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 38–44, 2015. doi: 10.1109/SPRO.2015.15.

[13] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: A multi-level anomaly detector for android malware. In *Mathematical Methods, Models, and Architectures for Network Security Systems*, 2012.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), jun 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL `https://doi.org/10.1145/2619091`.

[15] H.H. Feng, O.M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003.*, pages 62–75, 2003. doi: 10.1109/SECPRI.2003.1199328.

[16] H.H. Feng, J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 194–208, 2004. doi: 10.1109/SECPRI.2004.1301324.

[17] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7417-2. URL `http://dl.acm.org/citation.cfm?id=525080.884258`.

[18] Chris Frohoff. Appseccali 2015: Marshalling pickles - how deserializing objects will ruin your day. `https://frohoff.github.io/appseccali-marshalling-pickles/`, 2015. Accessed: 14-02-2023.

[19] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 318–329, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6. doi: 10.1145/1030083.1030126. URL `http://doi.acm.org/10.1145/1030083.1030126`.

[20] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. Behavioral distance measurement using hidden markov models. In *International Symposium on Recent Advances in Intrusion Detection*, 2006.

[21] Jonathon Giffin, Somesh Jha, and Barton Miller. Efficient context-sensitive intrusion detection. 12 2003.

[22] Google. Libunwindstack git repository. `https://cs.android.com/android/platform/superproject/+/master:system/unwinding/libunwindstack`, 2023. Accessed: 13-01-2023.

[23] R. Gopalakrishna, E.H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 18–31, 2005. doi: 10.1109/SP.2005.1.

[24] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[25] Lok Kwong and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. *Proceedings of the 21st USENIX Security Symposium*, 01 2012.

[26] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. 7, 02 1998.

[27] Zhen Liu, S.M. Bridges, and R.B. Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Third IEEE International Workshop on Information Assurance (IWIA'05)*, pages 164–177, 2005. doi: 10.1109/IWIA.2005.6.

[28] Nuno Nogueira. *Remote Debugging Detection on Android*. PhD thesis, Università Ca' Foscari Venezia, 2022.

[29] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 144–155, 2001. doi: 10.1109/SECPRI.2001.924295.

[30] Roman Shrestha. *Remote Debugging Detection on Android*. PhD thesis, University of East London, 2018.

[31] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. 01 2015. doi: 10.14722/ndss.2015.23145.

[32] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bruno Kessler, Bert Abrath, and Bart Coppens. Reactive attestation: Automatic detection and reaction to software tampering attacks. 10 2016. doi: 10.1145/2995306.2995315.

[33] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 156–168, 2001. doi: 10.1109/SECPRI.2001.924296.

[34] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, page 137–148, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311595. doi: 10.1145/2348543.2348563. URL https://doi.org/10.1145/2348543.2348563.

[35] Kui Xu, Danfeng Daphne Yao, Barbara G. Ryder, and Ke Tian. Probabilistic program modeling for high-precision anomaly classification. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 497–511, 2015. doi: 10.1109/CSF.2015.37.

[36] Lifan Xu, Dongping Zhang, Marco A. Alvarez, Jose Andre Morales, Xudong Ma, and John Cavazos. Dynamic android malware classification using graph-based representations. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 220–231, 2016. doi: 10.1109/CSCloud.2016.27.

[37] Rubin Xu, Hassen Saïdi, and Ross J. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, 2012.

[38] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous path detection with hardware support. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005.

[39] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 611–622, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. doi: 10.1145/2508859.2516689. URL https://doi.org/10.1145/2508859.2516689.

[40] Min Zheng, Mingshen Sun, and John C.S. Lui. Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 128–133, 2014. doi: 10.1109/IWCMC.2014.6906344.