Detection of Reverse Engineering Activities before the Attack

Paolo Falcarin
Dip. di Scienze Ambientali, Informatica e Statistica
Ca' Foscari University of Venice
Venice, Italy
paolo.falcarin@unive.it

Mirco Venerba
Dip. di Scienze Ambientali, Informatica e Statistica
Ca' Foscari University of Venice
Venice, Italy
872653@stud.unive.it

Matteo De Giorgi
NATO Communications and Information Agency
NATO
The Hague, The Netherlands
matteo.degiorgi@ncia.nato.int

Federica Sarro
Computer Science Department
University College London
London, United Kingdom
f.sarro@ucl.ac.uk

***Abstract*—In typical cybersecurity scenarios, one aims at detecting attacks after the fact: in this work, we aim at applying an active defence, by detecting activities of attackers trying to analyse and reverse engineer the code of an Android app, before they will be able to perform an attack by tampering with the application code. We instrumented an app to collect various runtime data before and after deployment, in normal behaviour and under malicious analysis. We introduce the concept of partial execution paths as subsets of a program trace suddenly interrupted, as possible indicators of debugging activities. Such clues, along with system calls sequences and delays between them, stack information, and sensors data, are all data that are collected to help our system in deciding whether our app is under analysis and its device has to be considered compromised.**

***Index Terms*—Reverse Engineering, Mobile Apps Monitoring, Anomaly Detection, Software Protection, Anti-Piracy**

1. Introduction

Software piracy is a longstanding problem in software security: according to the European Union Intellectual Property Office [1], software piracy in 2017-2023 has steadily increased for mobile devices while in desktops it has slowly decreased. The creation of pirated software copies can be the result of various types of attacks: the 2025 Application Security Threat Report [2] shows an increasing number of environment attacks (when apps runs on rooted or jailbroken devices), instrumentation attacks (which involve dynamic code modification), and integrity attacks (where the app is modified and repackaged).
Such attacks falls into the wide category of Man-At-The-End (MATE) attacks [3], where the end user is the attacker, who could try understanding how the code works through the process of reverse engineering, copy and disclose without any authorisation codes in violation of copyright laws, modify and tamper with the code so that it behaves differently (such as removing license and payment checks), take advantage of some code vulnerabilities and much more. The typical MATE attack model consists of a sequence of three main types of activities, often in iterations, to achieve a particular malicious goal:

1) Static Analysis (using reverse engineering tools to analyse the app files)
2) Dynamic Analysis (typically debugging to collect runtime data from the app under analysis)
3) Code Tampering (modify the code to implement the attack)

Software protections against MATE attacks adopt different strategies, targeting the above activities to make it more difficult for the attackers: obfuscation makes static analysis harder for the attacker [4] often combined with software diversity [5] [6] to increase code variability; other runtime protections have been designed to thwart dynamic analysis with anti-debugging techniques [7] [8] while integrity verification has been used for tamper-detection [9]. More recently, such protections have been extended by the use of trusted servers for implementing tamper-detection with remote attestation [10], with mobile code [11] or coupled with native libraries [12]; trusted servers have also been used to renew and diversify Android app native code [13] to contrast dynamic analysis.

The goal of our work is designing a system able to stop an attacker before performing an attack, while they are still analysing our app code: we aim at monitoring, collecting, and then analyse and correlate multiple types of data gathered from an Android device during our app execution, in order to combine different clues that might indicate the presence of reverse engineering activities.

We assume that protected apps cannot be easily attacked with static analysis only, and that the attacker needs to run and debug our app before implementing an attack: at the same time, during its normal usage, our protected app will collect runtime data to be sent to a trusted server, where these device data and user behaviour are analysed to detect anomalies and unauthorised reverse engineering activities such as debugging.

This work was supported by project SERICS (PE00000014) under the National Recovery and Resilience Plan of the Italian Ministry of University and Research (MUR), funded by the European Union - NextGenerationEU.

In the next sections we will describe our proposed software architecture, list the collectable data that could hint about the presence of reverse engineering activities, describe our approach, and show an initial validation on a simple Android application.

2. Reverse Engineering Detection

Detecting reverse engineering activities require the installation of additional software libraries to collect data from the app runtime and from the device.

We have designed a software architecture (Fig. 1) that creates an interaction and protection mechanism between an Android application and a Trusted server. Information at different levels are collected from the device starting from low-level system calls up to multiple high-level data such as sensors, and device settings; then the trusted server will check these data to detect anomalous behaviour, or the presence of debugging, or other reverse engineering activities.

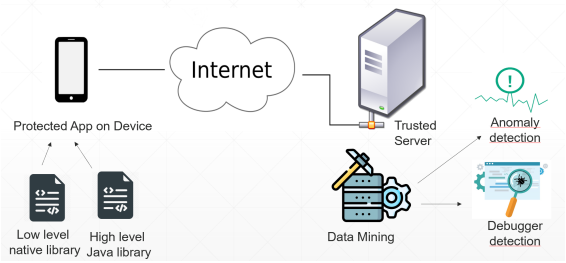

Figure 1. Architecture for Reverse Engineering Detection.

The whole process is composed of the following phases:

1) Running test-suites at development time for extracting data by means of the two libraries to monitor the app installed on the device: the native library extracts low-level information such as system calls, their sequences, and stack information, while the Java library for extracts data from sensors, settings, and configuration files;
2) Transmission to the Trusted Server of all these data extracted from the Android device, to be stored in a graph model for further processing;
3) After deployment, collecting data, comparing these with the stored model and decide which actions to perform in case of suspect behaviour: disconnection from the network, further monitoring, device blacklisting.
4) The trusted server can run an anomaly detection to detect differences in the configuration and sensors data of the device, and it can run the debugging detection to compare the runtime data with the pre-built model.

In Figure 2, it is possible to see the point of view that can be obtained using this native library on an Android application; in fact, it can 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, our native library 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.

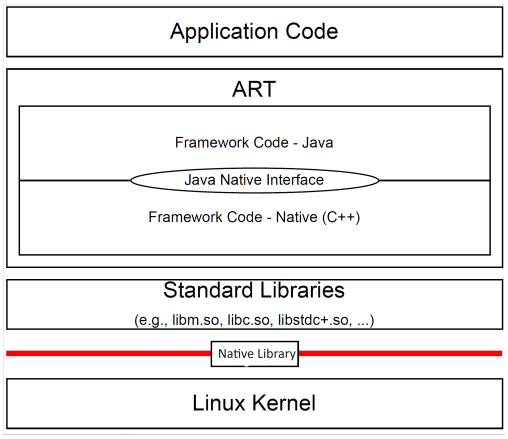

Figure 2. The native library in an Android environment

Mapping any interaction between the Android application and the underlying Linux kernel is a starting point to get a complete view of the actual behaviour of the application. We developed a native library for tracing any interaction of the protected app with the underlying Linux kernel: this low level library allows identifying the set of system calls used by the app, the various sequences of calls that form the program traces, and to collect the parts of stack data we are interested in monitoring.

We introduce the concept of *partial execution path* as a subset of a program trace suddenly interrupted: the appearance of such incomplete traces and corresponding delays between consequent system calls are a typical evidence of the activities of a debugger utilised to explore the app behaviour and collect runtime data from memory.

We also developed a Java high-level library, linked to the app, whose goal is to collect other data that might give away the presence of dynamic analysis by the user of the device.

2.1. Collecting Data from the App Runtime

We developed a native library for tracing any interaction of the protected app with the underlying Linux kernel: its main outputs are the set of runtime traces, where a single trace is a sequence of Linux system calls happening at runtime, and the stack memory contents of such calls. The `ptrace` interface in the Linux kernel enables a process ("tracer") to monitor and manage the execution of another process ("tracee"), and to inspect and modify the tracee's memory and registers. The tracee could also be a lightweight process, such as a thread.
This interface can be used via the system call `ptrace`, which allows one first to specify the process's identifier to trace and then how it should be traced. The identifier is generally known as the Process ID (PID), but it also accepts a Thread ID, which in Linux terms is often referred to as SPID. Moreover, it is essential to note that once a tracer is attached to a tracee, no other tracers for the same process (or thread) are allowed. In an Android environment, the usage of `ptrace` is further restricted in such a way that only the parent of a process can trace

it. Therefore, it is impossible to attach to any application, since they are always executed by the same parent. A way around this limitation is running the tracer process as root, allowing it to attach to any process. Alternatively, an application can trace itself by spawning a tracing process and explicitly allowing it to be a tracer.

Using 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 call parameters and return values.

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 utilized 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), which will need to be traced. To do that in the safest way possible, one can leverage on `ptrace`, which, thanks to some initialization options, allows one 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 they will all terminate if it comprises multiple threads, and only the thread group leader will be left. Such behaviour must be recognised, and all the data structures used to keep track of the system calls 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.

Using the `ptrace` interface implies being exposed to various special behaviours that Linux processes can assume and which need to be correctly addressed. For example, it is necessary to correctly handle `fork` family system calls and ensure that the generated child is traced, or correctly reflect the change that an `execve` family system call causes on a process (or thread family). To improve security, our native library will forcefully terminate the tracee, if the corresponding tracer process dies for any reason: 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 Analyser component. In Figure 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 will 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 is received when the kernel has performed the operation and allows the call's return value to be seen.

```
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.    0000000000000000
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 3. Partial output of tracing `ls` in Android without capturing the stack trace

For each system call, it is possible to read the stack trace that leads to its generation leveraging on the library `libunwind` [14] on generic Linux-based systems and `libunwindstack` [15] on Android. These libraries of-

fer 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.
Their main difference lies in the supported stack frame formats, which can vary greatly in Android systems. There are multiple ways to generate a backtrace, and they are heavily dependent on the used architecture: one basic way could consists of 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.

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 modern approach to stack unwinding leverages particular sections of the ELF file format (e.g., `.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 is 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 not to 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 most 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, which makes it essential to understand both. Moreover, there is an added complexity given by other stack frame formats like Chrome C++ frames, jit-ted 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.

The native library has been developed in C++ and targets the x86_64 and ARMv8-A (also called AArch64) architectures running Linux or Android: different flavours of the executable have been made for all four possible combinations, and various adaptations are applied to each one. This implies the project is compatible with the most modern Android physical devices (tested on Android 12), emulators, and any other Linux-based operating systems (tested on Linux kernel versions 6.x and 5.x) running on a supported CPU architecture.

2.2. Collecting Data from the Android Device

The high-level Java library is linked to the app code and it takes care of extracting high-level information from the Android device to be monitored, such as data from sensors, the state of the application in relation to the application lifecycle, any applications that have the debuggable flag set to true, any types of debuggers that can be connected while the user is using the device, and the type of recharge of the device. The first type of useful information to analyse to identify a possible debugger connected to an application installed on an Android device consists in the current state of the application to be traced in relation to its entire life cycle. We extract the periods of time in which the application is running and the periods of time in which the application is in the background (not running).

The main methods onResume and onPause can be traced (logging their start and end timestamps) for this purpose: the first method occurs when the activity becomes visible and can start interacting with the user, while the second method occurs when the activity is paused and this can precede putting it in the background or terminating its execution.

In the Android environment there are two types of debuggers: the first is the GDB debugger at native code level, and the JDWP debugger at app level. The main feature of the native library consists in the fact that it traces the execution of the application by connecting to its PID and consequently, knowing that in the Linux environment a process can only be traced by another process, we verify the name of the connected process for verify that it is the correct process and not the GDB debugger. The proc/pid/status file provides us with all the information of the process with that particular pid such as the name, the status, the connected process and much more. Consequently, a very important line is the one that begins with TracerPid which indicates the pid of the process connected with the current one. If this TracerPid is equal to zero it means that there is no connected process while if it is different, this indicates the pid of the connected process. Consequently the last step is to verify that the TracerPid is non-zero and if it is, access the proc/TracerPid/status file and extract its name. If this equals to our native library process we know that there is no GDB debugger attached. The *android.os.Debug.isDebuggerConnected* function can also be used to indicate whether the JDWP debugger is connected or not.

The *developer settings* contain also the *usb debug setting* with which a user could connect a generic computer and using the android debug bridge to access and modify device information, monitor and record screen content and install applications.

Another important condition to verify is the presence of a possible active connection between the device that the user is using and a computer under control by an attacker. To be able to do this, the type of USB charging can indicate that a connection has been established

between the device and a computer. In order to verify this mechanism we implemented a *BroadcastReceiver* that via the *onReceive* event, called whenever the event changes, saves the new state of battery charge of the device.

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.

We are also gathering data from as many sensors as possible to identify indicators that the running platform is not an actual user device. For example, if the motion sensor never detects any movement, not even when the user touches the screen, this can be considered a first hint that the platform might not be legit.

Sensors like the accelerometer, the gyroscope, the temperature, the position, the ambient light can give away some information that can be used to discriminate if our app running on the device is being used while in the hands of the user or while connected to a board to be remotely debugged with the help of a computer. If we assume that the attacker is debugging our application it means that the attacker himself is actively using a computer causing the device to remain in a stationary position over time. To identify this situation, we can use the data from the accelerometer, providing useful information about the acceleration on the three axes of the device, and from the gyroscope which provides information about the inclination of the device. To perform a more accurate analysis based on these two sensors, an advanced sensor called game rotation vector is used to put these data together by directly providing three output data: azimuth, pitch and roll which represent a measure in degrees in relation to the three axes of the device (See Fig. 4).

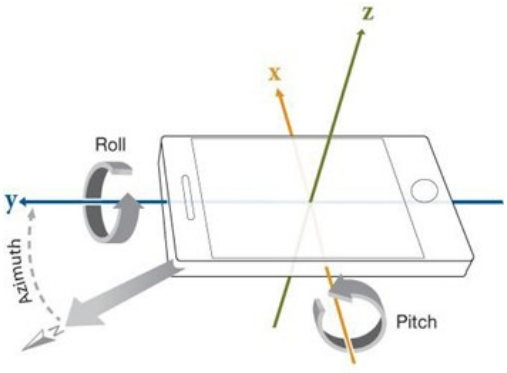

Figure 4. Rotation vector sensor

A fundamental point for the correct use and extraction of data from this sensor consists in the calibration, a process that is started a couple of seconds after starting the application, the time in which the user brings the device to a correct position and stable for the immediate following use of the application, and it is used to be able to subsequently identify invalid device positions: for example, when the phone is too much inclined for the user to actually use the app, or the screen is not visible from the user, or the phone is upside down.

All these data are sent to the trusted server that will mine these data to detect possible anomalies. All these hints, from sensors, traces and other configuration data, can be combined on the server and when multiple hints are raised, the trusted server can decide to consider a device as untrusted and take the appropriate measures, by stopping the app and related service.

In order to make this decision the Trusted server must be trained on the data so that it could actually check whether the data coming from one deployed app are producing too many hints of reverse engineering activities.

Depending on the training and the app, we can set the "security level" as the maximum threshold of hints that, when crossed, will determine that one device running our app has been compromised.

2.3. Ethical Implications

We are aware of the importance of transparency in communicating to users which data are being collected and why in our experiments. In order to address concerns about user privacy (according to GDPR article 4 [16]), we ask users to give their informed consent to use some of their personal data, namely: Android serial number, IP address, IMEI number, accelerometer sensor's data; we do not need data on location, contacts, or any other pictures or files in their phone.

3. Preliminary Evaluation

We developed a software architecture to collect any possible data from an Android device and the protected app's runtime data, measuring executions times, tracing system calls sequences and collecting sensors' data: all these data are sent to a trusted server. The trusted server checks on these data for making a decision on whether our protected app is under dynamic analysis from a malicious user. We applied our approach on two simple Android applications: a voice recorder and a calculator. The evaluation process consists of the following phases:

1) The training of the model created and used to identify a correct behaviour of an Android application through automated executions triggered by Macdroid: any different behaviour observed after deployment would be considered suspicious and raise an alert;
2) Normal executions performed by users to verify whether the libraries detect any incorrect / suspicious behaviour based on the data collected and analysed during the training phase and comparing them with the data collected during the last execution.
3) Correlating the anomalous executions, i.e. all those executions which are carried out in order to verify that, when various suspicious alerts occur during the current execution, the system functions correctly, i.e. detecting the alerts and mapping between security levels and each device are performed correctly.
4) Actions that can be taken when the system detects that a device has a high number of alerts: disconnecting the device from the network, uninstall the app, or continuous monitoring to confirm or refute the alert level.

The possible sequences of system calls collected before deployment are used as an oracle to detect anomalies with respect to the ones collected by our system after deployment. Such collected paths are used to build a graph where each path indicates an execution path monitored during the testing phase, before deployment. Consequently, a first point to be able to trace suspicious and non-suspicious executions consists in checking the current path in the graph and if this path is present then it means that it was traced during the training phase. As we cannot trace all the possible execution paths we must accept the possibility that some paths not traced at design time might appear after deployment whenever a user tries different inputs or app behaviours: such paths can raise alerts of suspicious behaviour that must be combined with other alerts to decide whether the device has been compromised. We used MacroDroid[1], an application for Android devices that automate a wide range of actions and processes with "macros", which are sequences of commands executed automatically in response to specific events or conditions. We used MacroDroid to create customisable macros to launch applications, press on the screen, access the file manager, save the execution results in some variables in an automated and repeatable way for collecting data before deployment: low-level data such as system calls with their sequences, characteristics, duration, common subsequences, incremental executions, and high-level data such as sensor data, developer settings, charging type, device stationary. We created ten macros with MacroDroid to describe valid execution paths and the same number for invalid execution paths. Each of these macros was executed twenty times to collect the following data on valid and invalid paths within the trusted server.

- Analysis of each method: minimum, maximum, average and variance time
- List of all execution paths to be able to observe any incremental or partial executions
- List of all possible subsequent system calls
- Time ranges in which a device is stationary
- Time ranges in which the device assumes invalid positions (such as the screen being upside down)
- If an already attached gdb or jdwp debugger is identified
- Applications with the debuggable flag in the manifest file enabled
- Time ranges in which the device is connected via usb to another device

For this preliminary evaluation, we defined the security level as the number of different types of alerts collected by the trusted server. We have defined ten types of alerts and we set at five the threshold to decide whether a device has been tampered with. In the current implementation all the alerts have the same weight but in a future versions it will be possible to consider alerts as features (with different weights) in a machine learning model for a more accurate calculation of the security level of that particular app. We tested our approach on two different Android applications: an audio recorder, and a simple open source calculator. We ran our two applications checking if the trusted server correctly identifies an execution that is different from the previous ones stored in our model. In Fig 5 we see the trusted server logs showing the present of alerts due some suspicious device configurations, while Fig 6 shows that the server as detected the presence of a Java debugger attached to our application.

1. https://www.macrodroid.com/

```
android - Connection from: #('192.168.1.3', 38662)#
ptracer - Connection from: #('192.168.1.3', 43918)#

Found debuggable application.
Security level of device = 1

Found developer options enabled
Security level of device = 2

Found USB charging type.
Security level of device = 3

Low level library not started
Security level of device = 10, device blocked
```

Figure 5. Suspicious Device Settings logged

```
ioctl has longer duration: 9449 vs 225->3346

gettimeofday has longer duration: 6729 vs 176->4058

read has longer duration: 9470 vs 183->8043

A jdwp debugger is found
Security level of device = 10, device blocked
```

Figure 6. Debugger JDWP found

4. Related Work

The OWASP foundation offers an extensive section in their Mobile Application Security Testing Guide (MASTG) [17] describing various Android Anti-Reversing Defences.
The guide proposes various solutions that can be composed together, forming a more extensive multi-layer approach to preventing and detecting reverse engineering attempts. We implemented some of the above-mentioned configuration checks: we control if the JDWP is connected by checking the Debuggable Flag, and we use *ptrace* to self-debug our application and prevent another debugging instance from running.

Self-debugging has been used to tightly couple a custom debugger to the application to protect and migrates code fragments to the debugger [7], effectively making dynamic analysis and reverse engineering harder, and preventing other debuggers from attaching to the application. In Android such approach has been extended by the Oblive project [8] by adding time-checks and hardware breakpoint checks, while Lim et al. [18] proposed a Java-based anti-debugging technique for Android and then a native library [19] checking the integrity of the Java-based protection, eventually detecting method hooking and code modification attacks by examining the call stack trace: in these cases the attacker may replace the entire native module and bypass the protection. Another Android debugger-detection was proposed by Wan et al. [20]: by using checkpoints for integrity verification, their system looked for the presence of open-source tools utilised to hook methods and APIs.
On the other hand, the novelties of our approach are manifold: the NFA representation of system calls for

monitoring executions and detect anomalies, the detection of partial execution paths, the use of sensors' data to detect mismatch between the device position and the app execution, and the use of a trusted server to analyse these data and decide if the device has been compromised.

System Call analysis has been extensively used by intrusion detection systems; Forrest et al. [21] proposed an n-gram model to validate small sequences of calls, while other techniques aimed at determining the normal behaviour of the program through static analysis [22], others using dynamic analysis [23] or a combination of both [24] to leverage on the advantages of both approaches. There has also been an evolution in the models used to capture the expected behaviour of an application; automaton transition verification was first described in [23], then formalized first as a Finite State Automaton (FSA) [25], and then as a Non-Deterministic Finite State Automaton (NFA) [22], showing that the call stack provides valuable information for detecting anomalies. The model has been further improved using Push Down Automata (PDA) leveraging on their stack to maintain the function call context [22].
All the previously mentioned models are based on static analysis. Most recent models such as Dyck [26], VPStatic [27], and the Inlined Automaton Model (IAM) [28], in the attempt to reduce the overhead of the PDA approach.

Other more advanced paradigms have been developed using a black-box approach to detect anomalies, similarly to our approach. Some notable examples are: VtPaths, which utilize return address information extracted from the call stack to build virtual paths [29], Execution Graphs, which provide a grey-box approach that accepts only system call sequences consistent with the program control flow graph [30], hidden Markov models where the hidden stochastic process consists in the aggregated tasks performed by the process (e.g., reading a file) and is observed by the emitted system calls [31], and its improved STILO model [32].

In the context of Android applications, similar models have been applied to System Call monitoring to detect anomalies as evidence of malware [33]–[35], where classification algorithms have been used to discern benign behaviours from malicious ones.
Other works aimed to provide a different level of insight on the actions performed by an application [36]–[38], which can help analyse malware. Various tools can acquire a similar level of detail on system calls as our native library , but their final goal is fundamentally different. For example, Perfetto [39] records the device activity for performance instrumentation and tracing analysis, and although it allows gathering a list of the invoked system calls together with their stack traces, it does not look for anomalies and it does not capture causal relationships in the model.
Our work provides a practical approach that specifically targets Android applications and aims to construct a behavioural model based on NFA and on Stack traces similar to VtPaths [29] and Execution Graphs [30]. Moreover, it aims to provide a deeper inspection of system calls to provide a view of the intended actions behind them (e.g., the interactions with the Android IPC Binder).

Other tools use a combination of static and dynamic analysis and leverage on the Linux process tracing interface `ptrace`; for example, DroidTrace [36] 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. Another relevant example is ProfileDroid [40]; 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, but it does not inspect Inter-Process Communications (IPC) as our native library does.

5. Conclusions and Future Work

We proposed a software architecture for detecting reverse engineering activities such as debugging for a client-server scenario, where the server needs to detect if their client apps have not been tampered with: such information can be used to identified a compromised device and its related user and stopping the service as a countermeasure. By intercepting and recording system calls, it is possible to gain a detailed understanding of the application's actions, identify unusual patterns of behaviour, and detect the presence of debuggers. Future developments will make the model suitable also to very complex applications, and apply machine learning to be able to predict with more accuracy when an attacker is performing reverse engineering activities on the device running our app. Future works will be also focused on merging the two libraries in a unique native library to be able to import all the functions in the same logical level and immediately above the kernel level in order to have immediate access to all the possible extracted data without any necessary modification of the code of the monitored application.

References

[1] European Union Intellectual Property Office (EUIPO), "Online copyright infringement in the european union – films, music, publications, software and tv (2017-2023)," https://www.euipo.europa.eu/en/publications/online-copyright-infringement-in-the-european-union-films-music-publications-software-and-tv-2017-2023, 2024, accessed : 15-4-2025.

[2] Digital.ai, "2025 application security threat report," https://digital.ai/products/application-security/, 2025, accessed : 15-4-2025.

[3] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. Khurram Khan, "Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions," *Journal of Network and Computer Applications*, vol. 48, pp. 44–57, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804514002379

[4] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[5] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[6] M. Ceccato, P. Falcarin, A. Cabutto, Y. W. Frezghi, and C.-A. Staicu, "Search based clustering for protecting software with diversified updates," in *Search Based Software Engineering*, F. Sarro and K. Deb, Eds. Cham: Springer International Publishing, 2016, pp. 159–175.

[7] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter, "Tightly-coupled self-debugging software protection," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/3015135.3015142

[8] D. Pizzolotto, S. Berlato, and M. Ceccato, "Mitigating debugger-based attacks to java applications with self-debugging," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3631971

[9] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Security and Privacy in Digital Rights Management*, T. Sander, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 160–175.

[10] S. Kumar, P. Eugster, and S. Santini, " Software-Based Remote Network Attestation ," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 05, pp. 2920–2933, Sep. 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TDSC.2021.3077993

[11] P. Falcarin, R. Scandariato, and M. Baldi, "Remote trust with aspect-oriented programming," in *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, vol. 1, 2006, pp. 6 pp.–458.

[12] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi, "Application-oriented trust in distributed computing," in *2008 Third International Conference on Availability, Reliability and Security*, 2008, pp. 434–439.

[13] B. Abrath, B. Coppens, J. V. D. Broeck, B. Wyseur, A. Cabutto, P. Falcarin, and B. D. Sutter, "Code renewability for native software protection," *ACM Trans. Priv. Secur.*, vol. 23, no. 4, aug 2020. [Online]. Available: https://doi.org/10.1145/3404891

[14] D. M.-T. Dave Watson, Arun Sharma, "Libunwind homepage," https://www.nongnu.org/libunwind/index.html, 2020, accessed : 2-3-2025.

[15] Google, "Libunwindstack git repository," https://cs.android.com/android/platform/superproject/+/master:system/unwinding/libunwindstack, 2023, accessed: 2-3-2025.

[16] European Union, "General Data Protection Regulation (GDPR), art. 4," https://gdpr.eu/article-4-definitions/, 2016, accessed : 10-4-2025.

[17] M. B. Carlos Holguera, Schleier Sven, *OWASP Mobile Application Security Testing Guide*, 1st ed., 09 2022. [Online]. Available: https://github.com/OWASP/owasp-mastg

[18] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, "An android application protection scheme against dynamic reverse engineering attacks." *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 7, no. 3, pp. 40–52, 2016.

[19] K. Lim, J. Jeong, S.-j. Cho, J. Choi, M. Park, S. Han, and S. Jhang, "An anti-reverse engineering technique using native code and obfuscator-llvm for android applications," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, ser. RACS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 217–221. [Online]. Available: https://doi.org/10.1145/3129676.3129708

[20] J. Wan, M. Zulkernine, and C. Liem, "A dynamic app anti-debugging approach on android art runtime," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2018, pp. 560–567.

[21] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, ser. SP '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–. [Online]. Available: http://dl.acm.org/citation.cfm?id=525080.884258

[22] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, 2001, pp. 156–168.

[23] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[24] Z. Liu, S. Bridges, and R. Vaughn, "Combining static analysis and dynamic learning to build accurate intrusion detection models," in *Third IEEE International Workshop on Information Assurance (IWIA'05)*, 2005, pp. 164–177.

[25] 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*, 2001, pp. 144–155.

[26] J. Giffin, S. Jha, and B. Miller, "Efficient context-sensitive intrusion detection," 12 2003.

[27] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 2004, pp. 194–208.

[28] R. Gopalakrishna, E. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," in *2005 IEEE Symposium on Security and Privacy (S&P'05)*, 2005, pp. 18–31.

[29] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy, 2003.*, 2003, pp. 62–75.

[30] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030126

[31] D. Gao, M. K. Reiter, and D. X. Song, "Behavioral distance measurement using hidden markov models," in *International Symposium on Recent Advances in Intrusion Detection*, 2006.

[32] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian, "Probabilistic program modeling for high-precision anomaly classification," in *2015 IEEE 28th Computer Security Foundations Symposium*, 2015, pp. 497–511.

[33] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for android malware," in *Mathematical Methods, Models, and Architectures for Network Security Systems*, 2012.

[34] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, and J. Cavazos, "Dynamic android malware classification using graph-based representations," in *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2016, pp. 220–231.

[35] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," 10 2011, pp. 15–26.

[36] M. Zheng, M. Sun, and J. C. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014, pp. 128–133.

[37] K. Tam, S. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," 01 2015.

[38] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 611–622. [Online]. Available: https://doi.org/10.1145/2508859.2516689

[39] Google, "Perfetto - system profiling, app tracing and trace analysis," https://perfetto.dev/, accessed: 2-3-2025.

[40] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of android applications," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ser. Mobicom '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 137–148. [Online]. Available: https://doi.org/10.1145/2348543.2348563