Responding to Stealthy Attacks on Android using timely-captured Memory Dumps

Jennifer Bellizzi

Supervisor: Dr Mark Vella

Co-supervisor: Dr Christian Colombo

February, 2024

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

L-Università ta' Malta
Faculty of Information & Communication Technology

L-Università ta' Malta

FACULTY/INSTITUTE/CENTRE/SCHOOL______________________
DECLARATION OF AUTHENTICITY FOR DOCTORAL STUDENTS

Student's Code _______________

Student's Name & Surname ___________________________

Course ____

Title of Dissertation/Thesis
____

____

(a) Authenticity of Thesis/Dissertation
I hereby declare that I am the legitimate author of this Thesis/Dissertation and that it is my original work.
No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education.
I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation and any other third party right infringement.

(b) Research Code of Practice and Ethics Review Procedure

I declare that I have abided by the University's Research Ethics Review Procedures. Research Ethics & Data Protection form code ____.

□ As a Ph.D. student, as per Regulation 66 of the Doctor of Philosophy Regulations, I accept that my thesis be made publicly available on the University of Malta Institutional Repository.

□ As a Doctor of Sacred Theology student, as per Regulation 17 (3) of the Doctor of Sacred Theology Regulations, I accept that my thesis be made publicly available on the University of Malta Institutional Repository.

□ As a Doctor of Music student, as per Regulation 26 (2) of the Doctor of Music Regulations, I accept that my dissertation be made publicly available on the University of Malta Institutional Repository.

□ As a Professional Doctorate student, as per Regulation 54 of the Professional Doctorate Regulations, I accept that my dissertation be made publicly available on the University of Malta Institutional Repository.

__________________	______________________
Signature of Student	Name in Full (in Caps)

____
Date

To my family

You are my all.

Acknowledgements

I would like to express my heartfelt gratitude to all those who have contributed to the completion of this thesis. Their support and constant encouragement were instrumental in my PhD experience.
First and foremost, I am indebted to my supervisors, Dr Mark Vella and Dr Christian Colombo. Their limitless patience, guidance and dedication were invaluable throughout the last four years. The amount of effort and time they dedicated to meetings, attentive feedback on numerous works, and funding efforts were impressive. Their input and contribution have shaped my approach towards research and academia. For this, I am immensely grateful; this thesis would not have been possible without you.
I also want to thank professors Julio Hernandez-Castro and Mauro Conti and assistant professor Eleonora Losiouk for their invaluable feedback and comments about the experimentation carried out as part of this thesis. Their feedback pushed the research in this thesis even further. Feedback from anonymous reviewers for paper submissions also contributed to the direction taken by this work.
A big thank you goes to the friends I met during this journey. Yonas, Kevin, Mark, Adrian and Duncan. Thank you for your support and encouragement and for always lending a helping hand whenever needed.
I would especially like to thank my family. My parents, who were always very supportive of any academic endeavour I set my eye on. For instilling in me a thirst for knowledge and always providing for me in whatever way I needed, thank you. As with all my previous endeavours, this would not have been possible without their patience, love and constant belief in me. Annalise, Reuben and Matthew, thank you for always being supportive, for always being there and having the right words to calm me down and cheer me up during particularly difficult times. I am lucky to be surrounded by you all as my family.
Last but not least, a huge thank you goes to Patrick. Whenever I thought some hurdle was too challenging, your unwavering belief in me always motivated me to push on. Thank you for your unconditional love, for always being by my side, and being so excited to proofread anything I write (many times). Your support has been invaluable, I dedicate this thesis to you.

Abstract

In recent years, several attack vectors have emerged which enable malware to hijack the functionality of targeted, benign apps. Some of these attack vectors have nearly been fully realised and give rise to a threat model where malware offloads key attack steps to the hijacked benign app functionality. In the process, attacks following this threat model evade malware detection that assumes malware to be self-contained. Moreover, through the same hijacked functionality, any attack traces can also be erased, rendering log-based attack investigation tools ineffective. This app hijack threat model needs anticipating through defensive measures before it manifests into an unmitigated threat.

Regardless of the stealthiness of an attack, any evidence must reside in volatile memory during its execution. However, collecting in-memory evidence associated with the app-specific hijacked functionality on Android devices is challenging. Current Android memory forensics methods for app analysis involve using devices which are custom or whose default security has been compromised. Moreover, randomly obtained memory dumps overlook the ephemeral nature of memory, which requires timely collection. Additionally, for the app hijack threat model, identifying app-specific artefacts in memory linked to hijacked functionality and extracting meaningful information from them necessitates an app-centric approach. This in-depth analysis of individual apps is infeasible and may require sacrificing default app protections.

This thesis aims to determine how attack steps offloaded to benign apps can be recovered from volatile memory in a timely and minimally invasive manner with respect to devices and apps. The proposed approach uses process memory introspection to collect real-time evidence from app memory, reducing reliance on app-specific logic. The study introduces Just-in-Time Memory Forensics (JIT-MF), a framework designed to explore this proposed approach within the constraints of stock Android devices and apps. JIT-MF consists of drivers that timely capture app-specific artefacts from memory through trigger points, a driver runtime supporting driver functionality, and produces JIT-MF logs containing app-specific evidence from memory.

The experiments conducted and described in this thesis demonstrate the feasibility of real-time app-specific evidence collection from the memory of Android stock devices using the JIT-MF framework. Results reveal that leveraging widely-used codebases for trigger point selection and app-specific artefact dumping avoids app and device-invasive methods while maintaining accuracy. JIT-MF trigger-based memory dumping improves state-of-the-practice by producing forensic timeline sequences that accurately reconstruct app-specific attack steps for this threat model.

Contents

List of Figures

List of Tables

List of Publications

Jennifer Bellizzi, Mark Vella, Christian Colombo, Julio-Hernandez Castro (2021) "*Real-Time Triggering of Android Memory Dumps for Stealthy Attack Investigation*" NordSec 2020; Lecture Notes in Computer Science, vol 12556. Springer, Cham.
Contribution: Primary author and independently completed the first draft. Independently developed the tool prototypes and automation scripts as well as environment setup required for experimentation. Experiment design, compilation and analysis of results was conducted under supervision.

Jennifer Bellizzi, Mark Vella, Christian Colombo, Julio-Hernandez Castro (2021) "*Responding to Living-Off-the-Land Tactics using Just-In-Time Memory Forensics (JIT-MF) for Android*". In Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT) SciTePress, pages 356-369.
Contribution: Primary author and independently completed the first draft. Independently developed the tool prototypes and automation scripts as well as environment setup required for experimentation. Experiment design, compilation and analysis of results was conducted under supervision.

Jennifer Bellizzi, Mark Vella, Christian Colombo, Julio-Hernandez Castro (2022) "*Responding to Targeted Stealthy Attacks on Android Using Timely-Captured Memory Dumps.*". In IEEE Access, vol. 10, pages 35172-35218.
Contribution: Primary author and independently completed the first draft. Independently developed the tool prototypes and automation scripts as well as environment setup required for experimentation. Experiment design, compilation and analysis of results was conducted under supervision.

Jennifer Bellizzi, Mark Vella, Christian Colombo, Julio-Hernandez Castro (2023) "*Using infrastructure-based agents to enhance forensic logging of third-party applications*". In Proceedings of the 9th International Conference on Information Systems Security and Privacy,

(ICISSP) SciTePress, pages 389-401.
Contribution: Primary author and independently completed the first draft. Independently developed the tool prototypes and automation scripts as well as environment setup required for experimentation. Experiment design, compilation and analysis of results was conducted under supervision.

Jennifer Bellizzi, Mauro Conti, Eleonora Louisouk, Christian Colombo, Mark Vella (2023) "*VEDRANDO: A novel way to reveal stealthy attack steps on Android through Memory Forensics*". In Journal of Cybersecurity and Privacy. 2023; 3(3):364-395.
Contribution: Primary author and independently completed the first draft. Independently developed the tool prototypes and automation scripts as well as environment setup required for experimentation. Experiment design, compilation and analysis of results was conducted under supervision.

1 Introduction

Android has established itself as a leader in the mobile OS market [84], making devices a rich source of evidence and their users a primary target for cyberattacks. Several protection and detection mechanisms exist in the Google Play Protect suite to hinder the availability of malicious apps on the app store, complemented with on-device detection [37]. However, the attack vectors malware developers use are becoming increasingly sophisticated, aiming to render malware stealthy enough to evade such detection.

A group of attack vectors have emerged in recent years, which have been shown to have the potential to launch stealthy attacks by hijacking benign applications [71, 148, 110, 137]. Android's accessibility feature is a case in point [49]. While rendering every app on an Android device accessible to alternative means of interaction, accessibility services also make it possible to backdoor the Android app containerisation security model. This attack vector has been leveraged to hijack messaging conversations and fund withdrawals through cryptocurrency wallet theft, bypassing two-factor authentication [70].

Such attack vectors elicit the app hijack threat model, which comprises malware that offloads attack steps to legitimate benign app functionality. The same benign app functionality can also be hijacked to hide or delete any traces of the executed attack steps, thus remaining undetected. These attack vectors have been observed heavily in malware for other purposes, including coercing the victim into clicking an unexpected prompt or identifying when a sensitive app is in use. Yet, some have also been used to nearly fully realise the app hijack threat model, as observed in the WhatsApp Pink worm [7], which hijacks the message-sending functionality of installed benign instant messaging apps. Consequences of such attacks range from personal information leakage and safety to financial theft, depending on the hijacked app.

The attack steps executed by malware following the app hijack threat model are carried out by hijacked benign app functionality. Therefore, typical malware detection at the app store and on the device leveraging behaviour-based threat detection or assuming malware to be self-contained [77, 80, 117] is rendered useless. When detection mechanisms fail [66], an alert is raised after the consequences of the attack become evident to the victim. The onus then falls on incident response to recover and remediate attack steps. Current incident response tools, including mobile forensics tools, still rely heavily on stored data and logs belonging to third-party applications to collect evidence of attack steps [76, 101, 31].

In the case of attacks following the app hijack threat model, evidence of attack steps constitutes the targeted benign app functionality hijacked, resulting in the targeted app's logs as the primary source of forensic evidence. Therefore, investigators are at the mercy of the hijacked app's default logging abilities. Even if the logs contain the necessary evidence for attack step recovery, previous research has shown that the same threat model can be used to erase evidence of attack steps [70, 130], depending on the functionality exposed by the app.

Regardless of the stealthiness of an attack, its execution must occur in volatile memory [27, 26]. However, memory forensics presents unique challenges on Android when collecting evidence for attacks following the app hijack threat model. Firstly, memory acquisition functionality is unavailable on stock Android devices. Additionally, the ephemeral nature of volatile memory and the stealthiness of the app hijack threat model signify that any artefacts in memory are short-lived and app-specific. Thus requiring timely collection, ensuring any relevant evidence can be caught whilst still in memory and app-specific knowledge.

1.1 Motivation

State-of-the-practice presents a gap. Namely, arbitrarily-timed memory dumps fail to address the challenge of timely artefact acquisition from memory, required due to the ephemeral nature of the stealthy attack's evidence in memory. Furthermore, current Android memory acquisition approaches are invasive; they require modifying the manufacturer's stock device or third-party apps [119, 146, 147]. Installing customised Android kernels that enable taking memory dumps requires device rooting, an irreversible process unique to device manufacturers, which weakens the device's default security mechanisms that typically ensure apps operate with minimal privileges. This is a concern if the user continues using the device after memory acquisition, as in an incident response setting. Compatibility with stock apps is also required for similar reasons. An invasive approach also concerns the understanding of app internals. Mapping app-specific functionality to hijacked events as attack steps requires an in-depth analysis of the app comprising app disassembling and compiled code analysis [65, 153]. Otherwise, only objects containing character strings and primitive types would be retrievable [4, 154].

Any app-specific approach is considered operationally infeasible due to the sheer number of apps available in app stores and may introduce new concerns regarding the stability of the hijacked app.

Android memory introspection is already used for app usage reconstruction [19, 102] and extracting specific objects like TLS keys for network forensics [122]. It can also be leveraged in sandboxed environments to detect malicious apps before uploading them to the app store [5, 120]. Nonetheless, some malware evades detection, requiring incident response to reconstruct and remediate the attack steps. The incident response cycle calls for a context that bears the device user in mind. Typical Android users use stock devices, as shipped by the manufacturer. However, stock Android devices do not expose the functionality for dumping the contents of volatile memory, thus hindering memory introspection and analysis. Rather arbitrarily-timed memory dumps must be taken from Android devices with unlocked bootloaders and customised kernels installed [88, 48]. Further challenges related to the collection of app-specific attack steps involved in app hijacking attacks include determining which app-specific objects in memory can be linked to hijacked functionality and extracting object information related to the hijacked app functionality that is meaningful.

1.2 Research overview

The current limitations of the existing forensic tools and the gaps present by the state-of-the-practice in Android memory forensics in the context of the app hijack threat model described present a research question. **How can attack steps offloaded to hijacked app functionality be timely recovered from volatile memory of stock Android devices, in the least invasive way possible concerning both the device and the hijacked app?** The central hypothesis put forward is that timely and minimally invasive attack step recovery from volatile memory is possible by focusing on the technology stack layers below the application layer while still producing accurate forensic timelines as when operating directly on the application layer.

Just-in-Time Memory Forensics (JIT-MF) is a novel conceptual framework that is proposed in this thesis, for which implementation was made available, to explore the underlying hypothesis of this work. It leverages process memory introspection to conduct real-time memory forensics. The main components of this framework are: i) Evidence objects: app-specific artefacts whose presence in memory implies the execution of some specific app functionality, possibly a delegated attack step; ii) Trigger points: process instructions associated with the presence of evidence objects in memory and hence when memory dumps should be triggered; and iii) JIT-MF logs: the resulting logs JIT-MF produces that contain evidence objects comprising app-specific artefacts, timely dumped from memory. Trigger points and evidence objects are defined in JIT-MF Drivers that drive the real-time collection of app-specific artefacts from memory and which produce JIT-MF logs. The JIT-MF Driver Runtime provides all services the JIT-MF driver requires to operate, specifically app instrumentation, memory introspection and persistence of evidence to storage.

The concept of the JIT-MF Driver and Runtime, may be implemented at different layers within the Android technology stack. The layers in the stack range from the app-specific layer at the highest layer to the Linux kernel at the lowest. Yet implementing JIT-MF within the different layers of the technology stack calls for varying degrees of device and app-invasive approaches. Leveraging app-specific layers requires in-depth knowledge of the hijacked app, whereas lower levels avoid this requirement and, therefore, are less app-invasive. Conversely, leveraging services found in the lowest levels of the stack requires knowledge and modification of device-specific functionality, which results in a device-invasive solution.

1.3 Aims and objectives

This thesis aims to enable investigators to respond to this stealthy Android app hijack threat model. While protection and detection mechanisms can aid in minimising the possibility of attacks occurring in the first place, persistent attackers can bypass these mechanisms through the app hijack threat model that hijacks benign app functionality. Therefore, incident response is necessary to ensure any attack steps can be reconstructed so they may be remediated. An optimal solution can faithfully reconstruct the attack steps timeline, even if the stealthy malware has erased these steps and their evidence is short-lived in memory. The research question and hypothesis put forward call for the following objectives to be met by the research carried out as part of this thesis:

O1 Describe the app hijack threat model and understand how it exposes the limitations of existing countermeasures.

O2 Determine how a minimally invasive approach affects the ability to collect necessary app-specific artefacts from memory, linked to attack steps.

O3 Demonstrate how the collected evidence from memory comprising hijacked app-specific artefacts is critical for reconstructing attack steps for the app hijack threat model.

1.4 Contributions

This thesis aims to meet the objectives defined in the previous section by proposing a JIT-MF framework for the timely collection of attack steps from the volatile memory of hijacked apps. Subsequently, through multiple attempts at JIT-MF tool realisation, this thesis explores which layer of the Android technology stack is ideal for minimal invasiveness while collecting the necessary app-specific artefacts from memory. Thus, the main contributions of this work are the following:

- Trigger-based memory dumping can produce accurate forensic timeline sequences with sufficient detail one would normally expect only from developer-provided app logs.
- An exploration using an implementation of JIT-MF showed that by leveraging lower layers of the technology stack, app-specific artefacts could be collected while simultaneously being less app and device-invasive.
- The app-specific elusive evidence from memory found inside JIT-MF logs improves the state-of-the-practice incident response tools in the case of app hijack attacks.

1.5 Document structure

This thesis is organised as follows. Chapter 2 presents a literature review and background covering the current state of Android malware detection and incident response. The chapter covers existing techniques and applications for artefact collection from Android memory. Insight is given into the methods of Android app instrumentation, which is the core enabler for JIT-MF real-time evidence collection. The chapter also shows what invasiveness looks like across the Android stack and presents the Android security model with corresponding attack surfaces, highlighting the potential for stealthy Android malware developers. Chapter 3 describes the app hijack threat model and how attacks that follow it can evade detection, thereby remaining stealthy. Given the motivation, limitations of the state-of-the-art and potential benefit of memory as a forensic source, Chapter 4 presents the core components of the Just-in-Time Memory Forensics framework. Chapter 5 presents experimentation regarding the level of invasiveness needed for selecting trigger points and app-specific artefacts from memory that produce accurate evidence of attack steps from hijacked benign app functionality. Chapter 6 presents the results from evaluating the impact of using JIT-MF logs as an additional forensic source when using existing state-of-the-art forensic tools. Chapter 7 and Chapter 8 conclude this thesis by presenting a further discussion on experimentation and results, as well as conclusions and directions for further research.

2 Android malware detection & response

This chapter describes existing detection mechanisms aiming to detect malicious apps and activity in Android (Section 2.1), whose limitations are uncovered in the case of evasive malware. These limitations require effective incident response practices and tools to remediate evasive attack steps (Section 2.2). However, current incident response tools fall short in collecting the necessary evidence to reconstruct attack steps of stealthy attacks that hide their forensic footprints. These limitations present a gap, which can be addressed through memory forensics (Section 2.3). Current Android memory forensics research partially addresses these limitations. Yet these works do not consider the requirements of an incident response setting or the timeliness of acquisition from memory required to capture evidence of stealthy attacks that go undetected for a longer duration, such as app hijack attacks. The timely capture of artefacts in memory remains a challenge, which has only been addressed by few works focusing solely on carving objects in memory on Android and virtual machine introspection, for which enabling technologies are relevant (Section 2.4). Limitations regarding device and app-invasive approaches (Section 2.5) related to Android memory forensics as part of incident response, as well as possible enabling technologies for timely collection from memory, remain unaddressed. This is a cause for concern for Android users in the face of the attack surface that the Android security model exposes (Section 2.6).

2.1 Android malware detection

The primary protection against malware is early-stage detection; that is, malware prevention. Several detection mechanisms exist in the Google Play Protect suite,[1] both within the Google Play Store to hinder the availability of malicious apps as well as to

[1] `https://developers.google.com/android/play-protect`

provide on-device detection. Yet the increasing sophistication of Android malware to evade detection presents a significant challenge for early-stage detection mechanisms to keep pace. Between January 2016 and July 2021, 1,238 malware samples were found to have penetrated the Google Play app store [25]. Even more concerning is that once such malware is made available on a recognised platform, it quickly reaches Android users' phones. Fake versions of legitimate apps, banking trojans and malware bundled up in benign-looking apps, near millions of downloads from Google Play Store itself [92, 91, 93]. So much so that recent studies show that in 2019, 67% of unwanted malware found running on victims' phones originated from the app store [66].

Typical Android malware detection offered by existing tools falls into three main categories: static detection, dynamic detection, and hybrid detection [74, 33]. Static detection relies on known malware patterns to detect the presence of malware. In contrast, dynamic and hybrid malware detection involve executing malware to some extent in an emulated environment that enables analysis of its execution. Yet malware authors actively develop evasion techniques to thwart these detection techniques.

Specifically for Android, other prevention methods include real-time permission checks that check for permission misuse. This is typically carried out to avoid leaking of personally identifiable information (PII) [42], ensuring apps access data with least privilege [156] and safeguarding against permission-abuse attacks [98, 136]. Yet, not all malicious apps require permissions classified as dangerous, to operate.

Mainstream malware detection tools are signature-based and can only detect malware when a comprehensive list of malware signatures is provided [142]. Studies have shown that by employing obfuscation techniques which modify the malware's pattern in some way, anti-malware tools leveraging signature-based detection are 48.69% less effective at detecting malware, thus allowing malware to evade detection [34]. Even with the adoption of machine learning frameworks for malware analysis, key feature selection still relies on known malware patterns and techniques to flag harmful behaviour to the user [77, 80, 117]

Dynamic code loading is an Android OS feature that enables benign Android applications to call another APK or malicious code to compile and execute it in real time. However, malware developers can use this feature to load their malicious codes dynamically, thus evading any static checks [94, 97]. While dynamic analysis can assist in detecting any malicious behaviour during the execution of malware, regardless of how the malicious code was loaded [97], there are still limitations in detection that malware developers can exploit. Stealthy malware that aims to evade detection can have features such as triggering malware activities only if certain conditions, such as checking if it is running in a real environment rather than an emulated environment (typically used for malware analysis) [143]. If the analysis time is limited to minimise resource consumption, the malware may not exhibit its malicious behaviour within the allotted time, resulting in false negatives.

The limitations of existing malware detection are even more evident in the case of elaborate stealthy malware behaviour, such as fileless malware, which aims to operate without leaving traditional traces or files on the victim's system. This can include memory-based execution, whereby the malware operates primarily in the compromised system's memory. Another approach involves using living-off-the-land tactics, which leverage legitimate system tools, processes, or scripts already on the device to execute malicious activities. In the same way rootkits aim to infect the OS to hide and conceal evidence of malicious behaviour, living-off-the-land tactics aim to weaponise OS functionality to attack the OS, guided by thinking about what a forensic analyst will interpret when looking at those events. The attacker is social engineering the analyst by creating malicious effects that look like normal activity on the system [90, 68].

The standard enterprise threat solution are Endpoint Detection and Response (EDR) tools that monitor and record events occurring on endpoints (devices, PCs, servers etc.) in real-time, providing security teams with the necessary visibility to investigate and mitigate threats. These tools provide advanced threat detection, investigation and response capabilities [2] typically by leveraging known malicious tactics, techniques and procedures (TTPs) or behavioural analytics to detect attack-related unusual behaviour. Such tools have become essential, especially on desktop and server endpoints, to monitor suspicious behaviour that evades detection using the above-mentioned methods. Yet implementing such solutions on mobile phones is more complicated. Deep visibility and monitoring require root permissions, meaning rooting the device in the case of Android phones. Furthermore, similarly to forensic analysts, EDRs can still classify stealthy malware leveraging living-off-the-land tactics as benign activity.

2.2 Android incident response

Once malware evades detection and successfully executes on a user's device, alerts are raised after the consequences of the attack become evident to the victim. Attack step recovery is then needed to remediate the effects of the malware. Incident response is the process of investigating a security incident to determine the root cause of the breach so that the vulnerability and threat are mitigated. Typically, incident response requires several forensic tools specific to the Android device (based on its manufacturer) where the incident occurred. The main aim of these tools is to collect all the evidence available on the device from varying forensic sources to generate forensic timelines that enable incident responders to analyse and detect possible attack steps.

2.2.1 Forensic sources

A critical step of the incident response process is identifying and collecting the necessary evidence from sources that contain evidence of attack steps. Regardless of the analytical capabilities of the tools used, attack steps can only be detected if forensic sources containing evidence of their occurrence are collected in the first place.

Android on-chip and removable flash memory constitute primary forensic sources for device and app events. System-wide sources can provide supplementary information about the underlying Linux kernel activities (via `dmesg`), system and device-wide app event logging (via `logcat`), user account audits, running services, device chipset info, cellular and Wi-Fi network activities (via `dumpsys`) [63]. Typically app-specific forensic sources reside both in internal storage (the `/data/data` sub-tree of the Android filesystem), which is accessible only to the app, and in external storage (`sdcard`). Internal storage houses app persistent files and cache, whereas external storage contains app media. Another forensic source typically associated with mobile devices is cloud storage. Given the large multimedia files handled by Android apps, combined with on-device storage constraints, cloud storage has become a popular medium for long-term storage, even used seamlessly by apps for regular operation and backups.

App data is increasingly being stored in encrypted form for security and privacy purposes (e.g. practically all mainstream messaging apps [12]). Beyond the app level, device-wide disk encryption has evolved across Android versions. Full disk encryption (FDE) has been replaced by file-based encryption (FBE) as of Android 10 [52], rendering it more practical and stable. For instance, the alarm clock works even if the screen is locked, and a full factory reset is no longer necessary, even if the device runs out of power before it shuts down properly. While providing users with an additional layer of privacy, FBE makes it more difficult for investigators to analyse forensic sources available on the device without using users' collaboration or exploits to decrypt the sources.

Collection methods and challenges Collecting forensic sources from Android devices can be done using logical or physical imaging [113]. Logical collection relies on the OS to parse the device file system from raw (non-volatile) flash memory content. There are multiple ways to acquire forensic sources from a device logically. Generic approaches include traversing Android's filesystem and collecting the necessary files (using `adb pull`) or using Android backups (available to devices as from Android 4.0+) via `adb` `backup`. This backup includes shared preferences files and files saved to an app's internal storage, external storage, and in the database directory [9]. On the other hand, physical imaging provides exact bit-for-bit copies of flash memory partitions and can be conducted purely at the hardware level (e.g. through JTAG). Both collection methods have to deal with Android's security barriers. For software-based collection, the barriers range from locked screens, password-protected cloud storage to custom backup formats and device rooting or downgrading the app to gain access to files stored in the app's internal storage [17]. While hardware-based/physical collections can bypass the above barriers, any form of physical imaging has to deal with FDE and FBE.

Collecting app-specific evidence from an app's internal storage requires device rooting. This relies on exploiting a kernel or firmware flashing protocol vulnerability [147, 113] or flashing a custom recovery partition[2] to add root-privileged utility. The latter may get further complicated by locked bootloaders. Restoring backups for forensic analysis depends on the type of backup taken [9]. Default automatic backups cover apps that run on Android 6.0 or later. Android preserves app data by uploading it to the user's Google Drive and protected by the user's Google account credentials.

Further still, apps can implement their custom `BackupAgent`, which excludes all of the app's data files from a typical Android backup, and backups are handled directly by the app. Backups are typically encrypted in the cloud or the device's external storage. While some apps give users the option to encrypt backups, store them on external storage and restore them using a key, other apps make use of the Google Drive backup method, which means that backups can only be accessed and restored via the implementation of custom handlers for the `onRestore()` API [10].

Forensic Analysis. The starting point for forensic analysis depends on what kind of collection is performed [63]. In the case of physical collection, it is first necessary to identify the file system concerned (typically EXT and YAFFS), to extract the individual files with possible decryption efforts. This first pass brings the evidence to a state equivalent to a logically acquired one. A typical analysis pass for Android constitutes SQLite file parsing, given its inherent Android support. From this point onward, the decoding of app evidence is highly app-specific.

Even after app-specific evidence is parsed, the application logs' content depends on the application developers, and may not contain the necessary evidence to contribute to forensic analysis. Application logs are the primary data source engineers, developers, and analysts use to diagnose issues in deployed applications. However, it is difficult

[2]`https://twrp.me/`

to know a priori where logs are needed to help diagnose problems that may occur in the future, especially ones that concern evolving malware threats [126]. Several works attempted to address this problem using different approaches, namely by inserting instrumentation at strategic locations within application code to identify known issues [150] or divergence from normal execution leading to application failure [151]. These works rely on the knowledge available before app runtime to diagnose known possible issues at application runtime. Yet, other challenges abound when requiring logs for diagnosing performance issues [126] or undesired output [155], for which no prior knowledge exists and hence require the ability to customise logging choices during runtime. Furthermore, the intentions of app developers concerning log entries may differ from those of investigators who look to logging for security concerns and application misuse. Simultaneously, exhaustively logging all application events that could be hijacked by default would incur significant overhead on all application users.

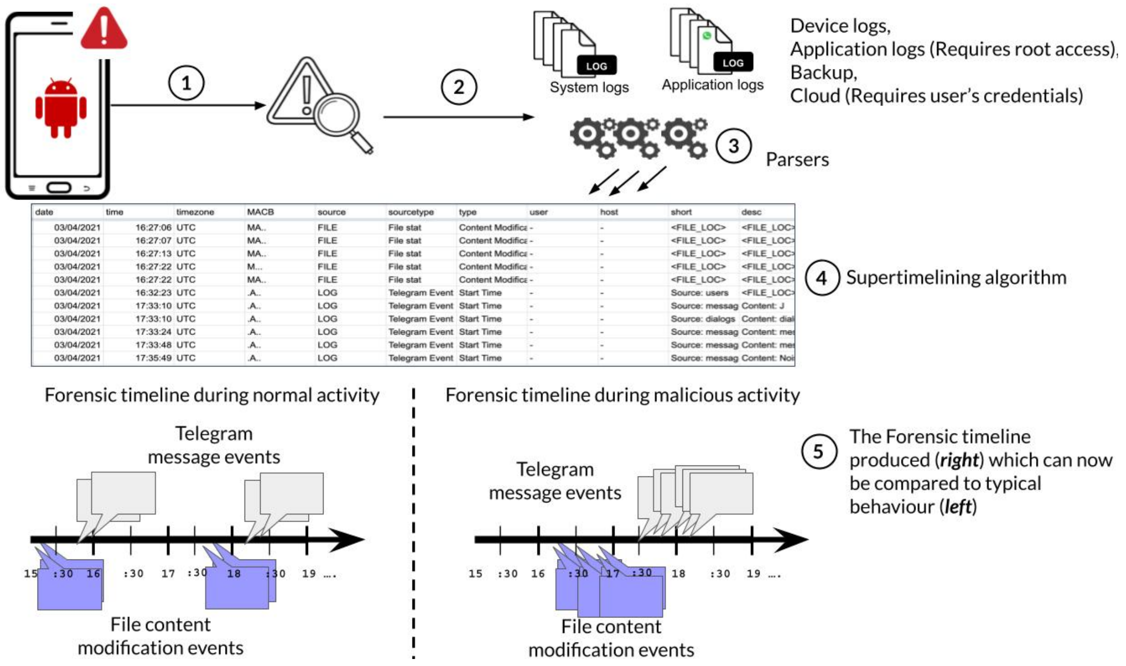
2.2.2 Forensic tools

Mobile forensics tools, e.g. Belkasoft [18] and MSAB's XRY [140], offer functionality for both the collection and the analysis of forensic sources from stock mobile devices. While each tool differs in how it collects evidence and analysis capabilities (including parsing of application logs), the forensic sources these tools collect are described in the previous section.

Each tool provides collection options with rooting exploits, hardware interfacing cables specific to different manufacturer devices and passcode brute-forcing methods. In some cases, app downgrading options are also provided to circumvent custom `BackupAgent` configurations that encrypt or omit app data from backups. They also have parsing/analysis modules for file systems, databases, and app data formats. Ancillary analysis features, including timeline generation, provide a final further aid to the investigator that eases the analysis process. The resulting evidence of interest can even be exported in a format as required by the investigator, typically CSV (Comma Separated Value) or JSON (JavaScript Object Notation) format.

2.2.3 Forensic timelines

Timeline analysis is one of the main forensic activities used to investigate attacks [40]. Evidence of attack steps can be dispersed across many forensic sources on the device. Forensic timeline generation is widely considered the forensic analysis exercise that combines all the collected evidence with the resulting forensic timeline presenting the


1
2
System logs
Application logs
Device logs,
Application logs (Requires root access),
Backup,
Cloud (Requires user's credentials)
3
Parsers
4
Supertimelining algorithm
Forensic timeline during normal activity
Forensic timeline during malicious activity
Telegram
message events
File content
modification events
5
The Forensic timeline produced (***right***) which can now be compared to typical behaviour (***left***)

date	time	timezone	MACB	source	sourcetype	type	user	host	short	desc
03/04/2021	16:27:06	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC
03/04/2021	16:27:07	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC
03/04/2021	16:27:13	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC
03/04/2021	16:27:22	UTC	M...	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC
03/04/2021	16:27:22	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC
03/04/2021	16:32:23	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: users	<FILE_LOC
03/04/2021	17:33:10	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: J
03/04/2021	17:33:10	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: dialogs	Content: dial
03/04/2021	17:33:24	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: me
03/04/2021	17:33:48	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: me
03/04/2021	17:35:49	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: Noi

Figure 2.1: The forensic timeline generation process starting with an incident alert raised by the device owner.

events before, during, and after a specific incident, providing the necessary context to reconstruct the incidents. Therefore, the richer the timelines, the greater the support for an investigator to reconstruct an intrusion/crime scene, answering critical questions about an incident [28, 59, 115].

Figure 2.1 shows how once an incident is flagged (step 1), an investigator initiates an investigation to recover and remediate attack steps by first collecting possible evidence (step 2) from forensic sources to generate a forensic timeline of events. These sources range from the device-wide `logcat` to app-specific sources inside `/data/data`, as well as inside removable storage, which can be found in the `sdcard` partition and whose mount point is device-specific (as described earlier in Section 2.2.1). Collecting some forensic sources typically also requires device rooting or a combination of hardware-based physical collection and content decryption. Following the collection of forensic sources, all collected artefacts coming from different sources are parsed and processed (step 3) to create a homogeneous series of events that can be inputted to a super timelining [56] algorithm (step 4) that condenses all events collected and produces the final timeline (on the right) (step 5).

When logs and evidence from different forensic sources have been collected, investigators are left with a list of events that occurred in the system, which need to be homogenised to create a single forensic timeline. The main technique for homogenising forensic sources to aid in forensic timeline reconstruction requires combining multiple "low-level events" comprising individual log entries from several forensic sources into fewer "high-level events" that are human-understandable events that investigators can use during analysis when reconstructing attack steps. Automated solutions that enable this process include event pattern matching [59], semantic-based correlation [28] and ontology-based techniques [29]. Regardless of the timeline generation method, the resulting events in forensic timelines depend on the available artefacts and forensic sources.

Forensic timeline tools Other than homogenising events from different forensic sources into a stream of events, further effort is required to parse artefacts from different sources into a format that can be ingested for forensic supertimelining. Plaso [56] is a Python-based engine equipped with a range of tools that can be used to create supertimelines, given logs from forensic sources automatically. Timestamped data collected from several sources can be parsed and chronologically ordered in one supertimeline using different Plaso parsers available for different application data. Furthermore, having publicly available source code, Plaso and its parsers can be extended to cater for additional forensic sources as needed. Timesketch [53] is another open-source tool that enables forensic analysis. It takes as input preprocessed timeline data (e.g. generated by Plaso tools) in JSON or CSV format and outputs a visual timeline (called a *sketch*) that can be analysed. Furthermore, like Plaso, it can produce a single sketch using multiple inputs of preprocessed timeline data containing parsed forensic evidence generated by various forensic analysis tools.

Identification of malicious events Generating forensic timelines is an important step during incident response, that aids investigators in determining the context for a set of events that occurred on the device. However, it is also crucial to identify events of interest within that timeline, especially in cases where a manual search within the forensic timeline does not narrow the set of events significantly, leading to a time-consuming and infeasible feat [115, 116, 135].

Machine learning techniques, namely anomaly detection, can be used to establish a baseline for normal activities in log files, which help identify those events within the timeline that could indicate suspicious activity (for instance, a comparison between left and right timelines in Figure 2.1 could indicate an anomaly) [115]. Several works [67, 73, 57] address challenges related to feature selection, automatic parsing of different, high-volume logs and finding the appropriate anomaly detection methods. However, the premise of such work is that the log events on which anomaly detection models operate include evidence of anomalous events.

2.3 Existing process memory forensics approaches on Android

Stored evidence can fall short of making all possible forensic artefacts available to generate a complete forensic timeline. This is especially true in the case of insufficient application logs [151] and stealthy attacks that evade detection and operate with a minimal forensic footprint [70]. As a result, recent approaches involve turning to memory as an additional forensic source of evidence [27].

Table 2.1 summarises the research directions for Android memory forensics regarding artefact reconstruction. These fall into two main categories i) Parsing of data structures from memory dumps through knowledge of Linux kernel data structures (under column "Kernel" in Table 2.1) or Android data structures through knowledge of Android heap memory manager data structures and those used by the system server (under column "Android" in Table 2.1); and ii) Carving of specific data structures from memory dumps based on expected data structure pattern. Parsing involves interpreting and extracting structured data based on predefined format rules, which in the case of kernel and Android data structures, comprise publicly-documented kernel and Android internals. Carving focuses on locating specific data structures in memory without explicit formatting information, using expected data patterns.

2.3.1 Memory profiling of kernel data structures

State-of-the-practice Android memory forensics relies on memory dumps. Popular memory forensics tools and frameworks, like Volatility and Rekall [132, 125] focus on analysing content in kernel space, which requires an entire memory image of the device. While tools like LiME [119] make retrieving such a memory image possible, using such tools on Android devices is challenging [27]. Root privileges are required on the device, and while this is typically easily achieved on systems via password access, mobile devices are by default unrooted; that is, rightful owners do not have root access as part of the safeguards that are in place on Android devices. Rooting a mobile device is an irreversible action that removes these safeguards. In the case of already rooted devices, LiME still requires an additional kernel module for collection to be inserted in the kernel, which typically involves recompiling a custom kernel.

Table 2.1: Summary of Android memory forensics research related to data structure parsing and carving from memory, along with their features. (marked in X).

Features	**Parsing of data structures in memory**		**Carving of data structures in memory**
	Kernel	Android	
	[132, 125, 86, 95]	[103, 104, 102, 4, 118]	[122]
Memory acquisition	X	X	X
Works with stock devices	X	X	-
Avoid device rooting	-	-	-
Avoid app modification	X	X	X
Triggered dump	-	X	X
App-specific artefact collection	-	-	X
Incident Response setting	X	X	X

Given that memory analysis is typically carried out on memory dumps, existing tools focus on improving kernel analysis [132, 95, 86] to recover and parse kernel data structures related to running processes, network sockets etc. However, these artefacts are limited when the objective is recovering app-specific data structures from memory. Memory analysis frameworks focusing on kernel data structure recovery require app-specific analysis plugins [27] that can parse kernel data structures to retrieve app-specific data structures. Therefore, if no app-specific analysis plugin is available, memory forensic analysis attempting to retrieve app-specific evidence from memory is restricted to retrieving character strings with no further context regarding the nature of the object within which these strings exist.

2.3.2 Parsing of Android data structures from memory dumps

Parsing of kernel data structures may help identify known malware that is present in the device's memory. However, evidence in memory can also be critical for providing necessary context during forensic analysis [27]. Data structures in memory dumps can be parsed to retrieve various object types; Primitives, Arrays, Strings, and Complex classes [4, 118], with varying effort required, depending on the object being parsed. Aiming to parse objects in memory related to apps' internal data structures can be effective in providing app-specific contexts such as app UI screens [102, 104] and photographic images [103], which could give forensic investigations further indication of app events. Yet this requires further knowledge of Android internals, including Android heap memory manager data structures and system server data structures.

Parsing Android-related data structures can produce more evidence to give more context during investigations than parsed kernel data structures. However, this type of evidence is still, in many cases app-agnostic [103, 4, 102]. While this is useful for producing contextual evidence from memory that can be used across apps, it still presents a limitation in the context of app hijack attacks. During investigations of such app hijack attacks, evidence of attack steps in memory comprises app-specific objects.

2.3.3 Carving of data structures from memory dumps

An alternative to parsing artefacts from memory dumps based on knowledge of Linux kernel data structures and data structures used by Android involves carving objects from unstructured memory dumps based on known object patterns. This avoids any memory parsing issues related to manufacturer-specific profiles [133], which may differ across varying manufacturer devices, while also being able to carve meaningful artefacts from memory. This, however, requires knowledge of the specific object that needs to be carved. Due to the specificity of the artefact to be carved, such approaches are limited in number. They are typically well-suited for specific scenarios when the pattern of the object to be retrieved can be predicted or has a specific pattern, for example, retrieving TLS master keys [122] and identifiable JSON message objects from memory [130].

Android process memory forensic research related to data structure parsing and carving assumes the presence of a memory dump. Such works, therefore, are not concerned with the timely acquisition of artefacts from memory due to their ephemerality. Moreover, they operate within a context that does not consider the implications of the device rooting process required to retrieve the memory dump.

2.3.4 Trigger-based memory dumping

The premise of collecting artefacts from memory is that memory contains critical evidence that can aid a forensic investigation of an incident. However, contents in volatile memory are ephemeral, meaning memory dumps need to be captured in a timely manner, that is, at specific instances when it is known that critical evidence is in memory. Otherwise, that evidence might go uncollected.

DroidKex [122] (the closest work to that proposed in this thesis) uses triggered Android memory dumps to ensure the presence of TLS keys that need to be carved within the acquired memory dump. Several other works that focus on the timeliness of memory dumps concern monitoring specific system events for which further analysis can be made through memory dumps triggered by that event. Most of the research in this area concerns virtual machine introspection (VMI) that enables the monitoring and detection of processes from a host machine [62] and addressing the semantic gap of VMI tools [82].

While the applications of hooking for timely memory introspection vary across different works [62], from malware detection to monitoring of connections and systems, the common theme comprises intercepting a third-party process's control flow through trigger events that indicate the presence of sought-after objects in memory. In the case of TLSKex [123], DroidKex [122] and SSHKex [108], this is done to monitor TLS and SSH communication of possibly malicious actors by retrieving TLS keys and SSH keys, respectively, from memory dumps. Therefore, triggers comprise function calls related to TLS and SSH initialisation that indicate the presence of respective master keys in memory. In these cases, once these functions are called, a memory dump is taken, and the sought-after objects in memory are carved from the memory dump, using known patterns and knowledge about the data structures used to store the SSH and TLS key data.

Triggered memory dumps have also been leveraged to enable rootkit detection [69, 32]. Since rootkits can stay dormant, and interfere with the system when needed, timely memory dumps triggered on the execution or events possibly related to rootkit activity (related to the change in control registers [32] or bus traffic activity [69]) ensure the memory dumps contain the evidence of possible rootkit activity. Hooktracer [26] uses the same premise and leverages the *SetWindowsHookEx* API as a trigger, to identify instances of fileless keylogger malware in memory.

2.3.5 Challenges of memory forensics

Memory forensics comprises both acquisition and analysis, which present several challenges within Android and in general. The main challenges regarding memory acquisition concern the: i) timeliness of acquisition [26]; ii) the availability of kernel profiles [85]; and iii) the reliability of the memory dump acquired [87, 27]. The ephemeral nature of evidence in memory requires timely acquisition, similar to the approach of works described in Section 2.3.4, which requires insight into the process for which evidence is collected.

The key component to dumping structured memory is kernel modules or drivers that enable the dumping and structured memory analysis, based on the running kernel. While this does not present an issue for Mac OS X and Windows, Linux kernel modules need to be compiled for every version of the kernel, of which there are several distributions [27, 87]. Android follows suit, however, with the additional challenges that different device manufacturers may have in their kernel customisations, making the process of maintaining and updating kernel modules for memory acquisition more complex. Even if automatic kernel profile generation can aid in generating appropriate kernel modules [85], Android still faces a unique problem whereby even installing the kernel modules responsible for dumping memory requires device rooting.

Page smearing is an inconsistency that occurs in memory captures when the acquired page tables reference physical pages whose contents changed during the acquisition process. As such, the acquired memory dump's reliability can be affected. Research works have addressed this by ensuring that the atomicity of a memory dump can be verified [27, 87]. Yet page smearing and potentially unreliable memory dumps present an issue for systems with over 8GB of RAM [27]. While increasing rapidly, the average RAM of Android devices is around 8GB. Therefore, page smearing does not present an immediate concern within the Android context.

The challenges related to memory forensics analysis of memory dumps comprise recovering evidence from memory to give the analyst necessary app-specific context [105]. Out-of-the-box memory analysis tools provide analysts with parsable kernel data structures, which require manual analysis by examining strings to retrieve meaningful app-specific information. This challenge applies to memory analysis in general and, therefore, within the context of memory dumps from Android apps. Previous work [105] aimed to tackle this using knowledge of internal Android data structures and program analysis to retrieve specific app information, specifically UI screens and photographic images. Yet, the evidence from memory required for further visibility during forensic analysis may call for retrieving other types of data not covered by this work.

2.4 Dynamic Binary Instrumentation for memory introspection

Dynamic binary instrumentation (DBI) is a candidate enabling technology for the timely acquisition of artefacts in memory, which can be introduced in the different layers of the Android technology stack. Binary instrumentation enables the insertion of code into an existing compiled binary to observe and/or modify the binary's behaviour. Instrumentation is typically used to gain insight into the application's performance at runtime regarding its behaviour, resource usage etc. DBI is a well-established technique [41] which offers a set of properties indispensable for program analysis. Namely, it can handle dynamically generated code. Setting up DBI requires two fundamental operations: i) Library injection: which allows a DBI library to be loaded inside the process to be instrumented, and ii) Execution interception: the functionality as part of the DBI library that intercepts the execution flow of the process [72]. Within the context of Android, instrumentation refers to modifying the behaviour of apps.

2.4.1 Application format and launch flow

Android apps are compiled and packaged as a single file, an APK (Android Package Kit) with `.apk` as the file extension. The typical makeup of an APK consists of an `AndroidManifest.xml` file providing essential information about the app, including permissions the app requests, a `lib` directory, and additional directories for resources such as images/files required by the app. An app can use both Dalvik bytecode found in `<classes>.dex` files and native code are compiled and stored in `*.so` binaries (shared object files) residing in the `lib` directory of the app, both of which can be unpacked to reveal the app source code.

Once an application is installed, it can be launched to run on the device in its process, through a series of steps that involve the following entities: i) an Activity component (i.e. an application component corresponding to an active interface on the screen) from which the user can request the launch of an application; ii) the Activity Manager Service at the Android API Framework layer; iii) a Zygote process, owned by root, the Android Runtime, and a set of functions, i.e. the Zygote Library (at the Libraries layer); and iv) a Zygote socket at the OS layer.

When a request is made to an app's Activity component to launch the app, via tapping on the app's icon, the Activity Manager Service is activated through the Binder IPC. The Activity Manager Service is responsible for issuing the necessary command (`Process.start()`) to the Zygote socket to start a new process. If the process had previously been started, the Activity Manager brings the application in question to the foreground. The `Process.start()` command connects the Activity Manager Service to the Zygote socket through a socket call, sending the Java class name whose static main method (as defined in the Activity component) will be invoked to specialise the child process associated with the app being launched.

The Zygote process, created on system startup, is the controlling process of the Zygote socket. When a command is received on the socket to start a new process for launching an app, the Zygote process invokes a fork system call at the Linux layer, to build a new process associated with the application being launched. Unlike in Linux systems, specialisation of the child process happens by loading the Java classes of the specific application inside the Dalvik VM or ART (Android Runtime), depending on the Android version, hosted in the child process rather than by loading a new executable image.

The newly-created process contains an instance of the Android `ActivityThread` class, which attempts to bind the process to the application, by loading the code corresponding to the main method inside the new process. If the creation of the new process and its binding to an application succeed, the Zygote process returns the PID to the Activity Manager Service, and the launch flow terminates successfully [15].

2.4.2 Library Injection

DBI's first step, library injection, involves injecting a library that allows the insertion of instrumentation code (the code that will modify the app's behaviour) into the intercepted code's process address space. This can be done in one of two ways: *statically*, or *dynamically* using system calls provided by the kernel such as `ptrace`.[3] The latter requires having elevated privileges on the device; in the case of mobile phones, this means that the device needs to be rooted, which is generally not the case and attaining such privileges involves exposing the mobile phone users to significant security risks. When elevated privileges are unavailable, static library injection becomes the only choice. This involves either inserting / modifying entries inside the app's Executable and Linkable Format (ELF) linker-related headers (in Android, this is the `.dynamic` section of the ELF file) [8] or patching the APK's bytecode [30] to include instructions that load the library when the process starts. Since these operations are performed statically on the executable, access to privileged system services is avoided. The library contains the hooking and instrumentation code which must be natively compiled for the required target architecture(s).

If the app to be instrumented already includes at least one native library, then the static manipulation of its ELF `.dynamic` section provides a viable option. Specifically, an additional entry is required to be added for the instrumentation library. Ultimately the ELF meta-data must not be broken. Had the `.dynamic` section expansion caused the follow-up sections to shift, then their pointers in the corresponding ELF headers must be adjusted accordingly. The only way to introduce a library on a non-rooted Android device is through an APK installation. That is, the hooking library must rather be included inside an APK. The entire process comprises: 1) Unpack the APK to be instrumented; 2) Modify the ELF header/`.dynamic` section of one of the existing libraries; 3) Add the instrumentation library to the app's *lib/* directory. Multiple versions for each required

[3]https://man7.org/linux/man-pages/man2/ptrace.2.html

architectural target must be included; 4) Repackage the app; and 5) Re-sign the APK (overriding the developer's original APK signature).

If, on the other hand, the app is not shipped with a native library, the APK's bytecode needs to be patched. The process would comprise the following steps: 1) Unpack the APK to be instrumented; 2) Identify the class within the app implementing the `MainActivity`, which executes when the app is launched, as listed in the app's Manifest file; 3) Statically patch its Smali code, that refers to the static constructor, to include the Smali equivalent of the function call: *'System.loadLibrary("library_name")'* that loads the instrumentation library; 4) Repackage the app; 5) Re-sign the APK.

Listings 2.1 and 2.2 show how a successfully updated `.dynamic` section looks like following library injection (`libhook.so`), using `readelf` (an ELF parser).

```
readelf -d ./libtmessages.30.so | grep NEEDED
0x0000000000000001 (NEEDED)        Shared library: [libjnigraphics.so]
0x0000000000000001 (NEEDED)        Shared library: [liblog.so]
0x0000000000000001 (NEEDED)        Shared library: [libz.so]
0x0000000000000001 (NEEDED)        Shared library: [libEGL.so]
0x0000000000000001 (NEEDED)        Shared library: [libGLESv2.so]
0x0000000000000001 (NEEDED)        Shared library: [libandroid.so]
0x0000000000000001 (NEEDED)        Shared library: [libOpenSLES.so]
0x0000000000000001 (NEEDED)        Shared library: [libdl.so]
0x0000000000000001 (NEEDED)        Shared library: [libc.so]
0x0000000000000001 (NEEDED)        Shared library: [libm.so]
```

Listing 2.1: `.dynamic` section of `libtmessages.so` (a library in the Telegram apk) prior to library injection.

```
readelf -d ./libtmessages.30.so | grep NEEDED
0x0000000000000001 (NEEDED)        Shared library: [libhook.so]
0x0000000000000001 (NEEDED)        Shared library: [libjnigraphics.so]
0x0000000000000001 (NEEDED)        Shared library: [liblog.so]
0x0000000000000001 (NEEDED)        Shared library: [libz.so]
0x0000000000000001 (NEEDED)        Shared library: [libEGL.so]
0x0000000000000001 (NEEDED)        Shared library: [libGLESv2.so]
0x0000000000000001 (NEEDED)        Shared library: [libandroid.so]
0x0000000000000001 (NEEDED)        Shared library: [libOpenSLES.so]
0x0000000000000001 (NEEDED)        Shared library: [libdl.so]
0x0000000000000001 (NEEDED)        Shared library: [libc.so]
0x0000000000000001 (NEEDED)        Shared library: [libm.so]
```

Listing 2.2: `.dynamic` section of `libtmessages.so` after injection librarylibhook.so.

While static injection of the library allows the insertion of instrumentation code in devices that do not make root privileges available, this comes with challenges. Anti-repackaging techniques may be employed to verify the integrity of the original app's codebase [78]. Such anti-repackaging techniques involve code integrity checks that operate both at the device and app levels that check for the presence of app repackaging techniques and codebase modifications.

2.4.3 Execution interception

Execution interception, the second component of DBI, allows control to redirect towards the instrumentation code and eventually back to the original execution. Interception can be carried out at various levels of granularity. At the most granular level, there is instruction-level interception by employing code cache-based methods [23]. This approach generally involves a just-in-time engine that performs dynamic compilation and instruments code one basic block at a time before executing, while storing instrumented code in code caches for faster access. This offers complete control over the instrumented process, which comes at a significant cost in runtime overheads. Less invasive approaches include bytecode manipulation and library function hooking [64]. Function hooking allows interceptions to be applied more selectively at runtime. This approach is less comprehensive and therefore has less control over the exact instructions being executed at the basic block level. It may even allow for the instrumented process to realise that it is instrumented [41], yet, it provides a practical real-time option.

GOT and PLT hooking. Since Android adopts Linux's dynamic linking method, library function/API hooking can follow the same Global Offset Table (GOT) and the Procedure Linkage Table (PLT) overwrite techniques [75] used in Linux. Specifically, to resolve the virtual addresses of imported functions from dynamically linked libraries (binding), the GOT and PLT are central to the whole process. Essentially, the GOT stores all the resolved library function virtual addresses, while PLT adds one further level of indirection to support lazy binding. Figure 2.2 shows how calls to library functions are routed through PLT instructions first. The call to the resolver (step 5) redirects execution to the link-loader (`ld`) if the `<addr>` in the GOT is not yet set. The link loader resolves the address of the required library function. Once resolved, the address is written to the corresponding GOT entry and the function is executed. GOT overwrites library function hooking by overwriting the resolved address (`<addr>`), with that of an address inside the injected library (e.g. `libhook.so`). The hooking routine should be implemented as a function designated to execute on library loading. PLT overwrites, on the other hand, achieves

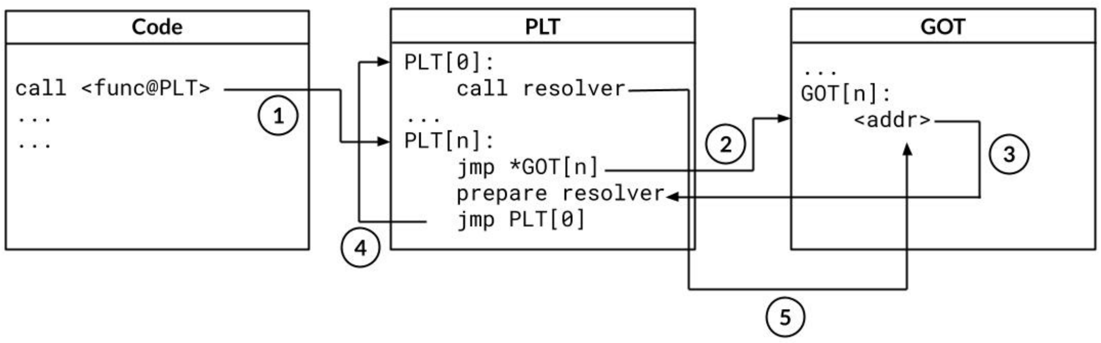


Figure 2.2: GOT and PLT library address resolution at runtime.

library function hooking by overwriting the GOT entry (`GOT[n]`), with that of an address in the GOT that points to the injected library (e.g. `libhook.so`).

Inline hooking. An alternative option is inline function hooking [22]. This approach disregards modifying ELF data structures and uses code patching directly. Inline hooking works by overwriting the first instructions inside the function to be intercepted with a jump (`jmp`) instruction to the instrumentation code. Apart from intercepting the code and executing the necessary command, the instrumentation code must then (1) include/handle any instructions that were 'lost' due to the instruction overwrite and (2) return execution flow to the original function. The latter steps are carried out by a function referred to as *trampoline*. This hooking technique modifies the instructions directly, making it more flexible than GOT and PLT hooking, which are limited to hooking only exported functions [8].

An Android app can use both Dalvik bytecode[4] and native code[5] (written in C or C++ and compiled into binary code). The Java API Framework and Java bindings are made available to Android developers. The former includes all the Android Java classes developers can use when building an application. The latter allows access to the functionality of native libraries. Since an app can run both native code and Dalvik bytecode, any form of inline hooking within the app must cater for both. Native code in the app runs directly on the CPU. It has access to the Android framework using Java Native Interface, which enables the switching between native code and Dalvik bytecode. Therefore, by employing an inline hooking approach that leverages the Android framework, all functions/methods in both native code and Dalvik bytecode can be intercepted.

[4]https://source.android.com/devices/tech/dalvik/dalvik-bytecode
[5]https://developer.android.com/ndk

2.4.4 Bytecode-level hooking

Xposed-style method hooking [139] is an inline hooking approach used to manipulate Android bytecode. It allows replacing the entire method body or introducing new code before and after the original Java method invocation within an app. Similarly to hooking done at the native runtime level, the Xposed-style method hooking uses a trampoline to jump to injected code and then revert to the original function. Unlike inline hooking, and similar to GOT/PLT hooking, given that the Android runtime operates on top of the native runtime and makes use of its data structures rather than rewriting the actual instructions of a function entry point, Xposed-style method hooking utilises the data structures and mechanisms provided by ART to achieve the same effect of the inline function hooking of native functions.

ART uses specific C++ classes to mirror Java classes, their methods and associated instances within a process, specifically using `Class`, `Object` and `ArtMethod` data structures. The `ArtMethod` data structure contains information about a particular Java method called *method descriptors*. These include the modifier (*access_flags*), the class in which it is declared (*declaring_Class*) and most importantly, the entry address of the method's code (*entry_point_**). The address stored here points to the address executed when the method/function is invoked. An `ArtMethod` instance can have either of four entry points *EntryPoint from Interpreter*, *EntryPoint from QuickCompiledCode*, *EntryPoint from JNI* and *EntryPoint from PortableCompiledCode* [83], depending on the value in *access_flags*.

Figure 2.3 demonstrates how Xposed-style method hooking leverages the `ARTMethod` data structure to hook specific functions. First, it patches the method modifier and sets it to `native` (`kAccNative`).[6] It then modifies the JNI entry point value (Step 2) to point to the address of the instrumentation code, and also stores in it the original backup values for the *access_flags* and *entry_point_**. The value of *entry_point_from_quick_compiled_code_*, which is the default entry point of a method ART follows at runtime, is modified (Step 3) to point to the address of the Xposed instrumentation handler (similar to the GOT/PLT trampoline). This handler obtains the address of the instrumentation code, executes it and returns the execution flow to the original function. Native functions executed during the app's runtime can be hooked using well-known techniques established on ARM architectures. These involve modifying a *trampoline-based hook* whereby the first few bytes of the function assembly code are replaced with a `jmp <hooking_function>` to the function containing instrumentation code [8].

[6]`https://android.googlesource.com/platform/art/+/master/runtime/art_method.h#118`

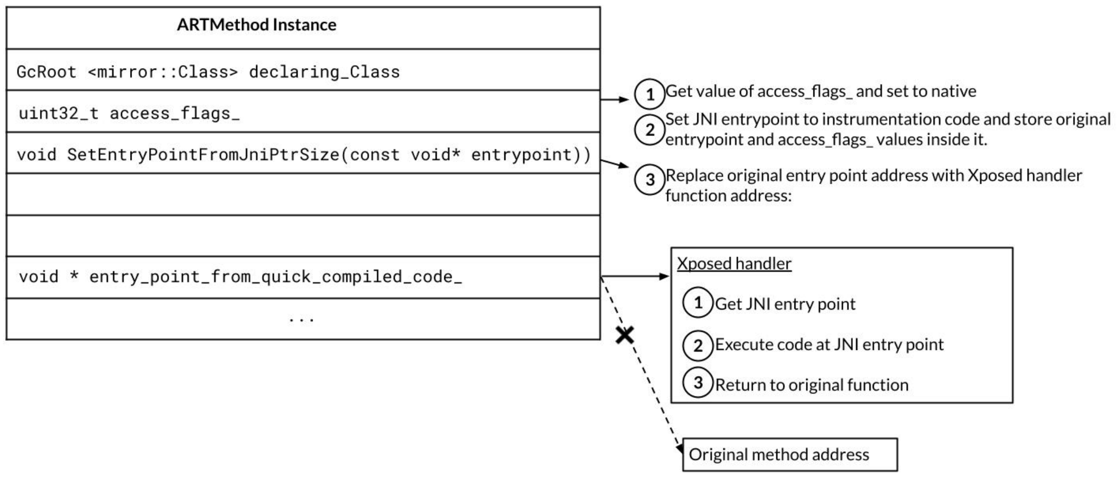

ARTMethod Instance
GcRoot <mirror::Class> declaring_Class
uint32_t access_flags_
void SetEntryPointFromJniPtrSize(const void* entrypoint))
void * entry_point_from_quick_compiled_code_
...
1 Get value of access_flags_ and set to native
2 Set JNI entrypoint to instrumentation code and store original entrypoint and access_flags_ values inside it.
3 Replace original entry point address with Xposed handler function address:
Xposed handler
1 Get JNI entry point
2 Execute code at JNI entry point
3 Return to original function
Original method address

Figure 2.3: Instrumenting code via `ArtMethod` entry point.

2.4.5 Dynamic runtime manipulation

DBI enables diverting the process execution flow through the instrumentation of program instructions. Yet changes to the app's behaviour can also be affected by manipulating its runtime environment; for instance, as used by app-level virtualisation frameworks [99, 3]. Android app-level virtualisation has emerged as a new technique that can load arbitrary third-party APKs. It enables an app (*container*) to create a virtual environment where other apps (*plugins*) can run. Plugins can execute independently from the underlying Android OS and other virtual environments. DroidPlugin [124] and VirtualApp [131] are the two most well-known frameworks supporting the generation of Android virtual environments and share a common design, as shown in Figure 2.4.

Figure 2.4 shows how the container app loads and runs plugin apps through a proxy. In this context, the app launch flow of a plugin app installed inside the app-level virtualisation environment is launched through the container app, which emulates the zygote process. The container app intercepts the Android API and inter-component function calls of the plugin apps, modifies the parameters, forwards them to the Android framework layer, and then intercepts and relays the responses back to the plugin apps, through the proxy. Meanwhile, the container app predefines stub components and permissions to cater for those required by plugin apps, and it encapsulates plugin app components in stub components at run time. In this way, multiple instances of the same app can bypass the UID restriction that disallows APKs with the same package name to have a different UID, and can now run simultaneously [111]. To distinguish between the different guest applications, the host application assigns them different process IDs. The proxy layer relies heavily on hooking mechanisms to communicate between Android

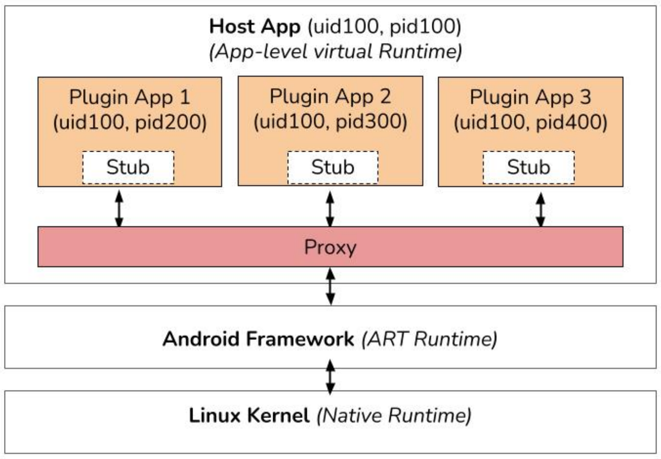

Host App (uid100, pid100)
(App-level virtual Runtime)
Plugin App 1 (uid100, pid200)
Stub
Plugin App 2 (uid100, pid300)
Stub
Plugin App 3 (uid100, pid400)
Stub
Proxy
Android Framework (ART Runtime)
Linux Kernel (Native Runtime)

Figure 2.4: App-level virtualisation architecture with container app managing the runtime of plugin apps in different processes but sharing the same unique user ID (UID).

system services and plugin apps. For example, it hooks `ClassLoader` to load plugin apps' DEX files and the inter-application communication (IPC) to manage and maintain the lifecycle of the plugin apps' components (such as starting and stopping an app Activity) [111]. Since the container app controls the launching flow of the plugin apps, the container app can also manipulate the process space of the plugin app by including, for instance, instrumentation libraries.

2.5 Minimal invasiveness of apps and devices

Similarly to memory forensics as carried out on desktops and servers, acquiring evidence from memory and other sensitive locations (i.e. app's internal storage) requires root privileges. However, unlike other systems, mobile phones present an additional challenge when requiring root privileges. The segregation of apps and their permissions is ingrained in Android's technology stack (see Figure 2.5). Therefore, root privilege on Android requires modifying the device or app in some way rather than simply requesting root permissions which can later be revoked.

Invasive acquisition approaches refer to collection methods that require app or device modification, some of which may weaken default security mechanisms. Significant modifications may be acceptable when dealing with virtual devices or confiscated devices that are no longer in use; for example, in post-mortem analysis or investigations involving perpetrator devices. However, this is not the case in the context of incident response aiming to remediate attack steps suffered by the victim.

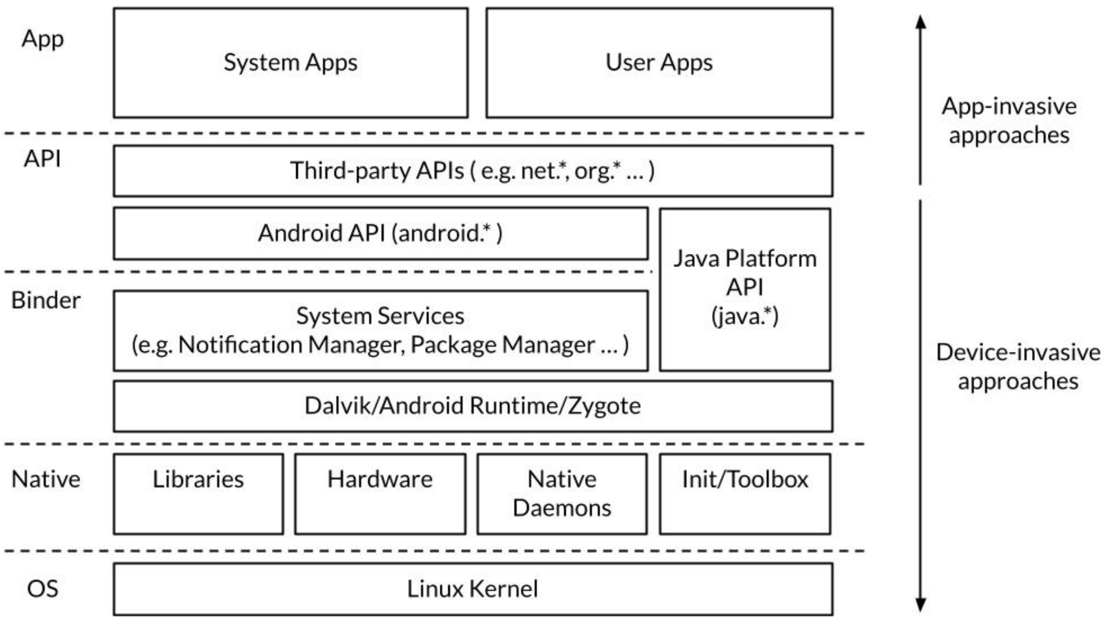

App
System Apps
User Apps
API
Third-party APIs (e.g. net.*, org.* ...)
Android API (android.*)
Java Platform API (java.*)
Binder
System Services (e.g. Notification Manager, Package Manager ...)
Dalvik/Android Runtime/Zygote
Native
Libraries
Hardware
Native Daemons
Init/Toolbox
OS
Linux Kernel
App-invasive approaches
Device-invasive approaches

Figure 2.5: Layers of the Android technology stack (based on the architecture presented in [44].

2.5.1 The Android technology stack

The modifications required for acquisition can occur in any of Android's technology stack layers shown in Figure 2.5. The stack consists of components that fall into five layers [43, 44]. Android applications fall into two categories: System apps and User apps. System apps are typically manufacturer-specific and come pre-installed in the system partition of stock devices, so they cannot be modified or uninstalled. These apps have elevated privileges and can access certain core functions of the operating system, yet this is not the same as having root privileges on the device. They are part of the device's firmware, and their updates are generally included in firmware updates provided by the device manufacturer or Android system updates. User apps are managed (installed, deleted and updated) by device users to extend the functionality of their devices. User apps operate within the boundaries of the Android security model and have limited access to system-level functions. They typically run in a sandboxed environment, with restricted permissions defined by the app developer and granted by the user. Unlike system apps, users have control over updating these apps, and they can receive updates through the respective app store or be set to update automatically.

The development of user apps relies heavily on well-maintained and widely-deployed underlying APIs that provide critical functionality through a well-documented interface. These comprise i) third-party APIs that provide specific functionality; ii) the Android framework libraries, which expose the functionality of various Android services; and iii) the Java runtime libraries on which the Android framework is built. Widely-deployed APIs are typically well-maintained, publicly available and offer backwards compatibility for specific functionality that may be required by the app, such as database functionality to store app state. Common third-party APIs include SQLite API[7] that manages the storage of app data and Firebase API that enables app development related to database communication, analytics and authentication.[8]

The Binder layer allows for communication between sandboxed apps and system services, depending on the permissions of the app granted to the app by the user. The native layer supports these layers. It contains the *init* binary components, which initiate all the processes in Android. The native component also contains native libraries used by the rest of the stack above. Some of these libraries may be common to all devices. Others may be manufacturer-specific, such as hardware-related libraries. Finally, at the lowest level of the stack is the Linux kernel, upon which Android is built. The kernel provides hardware, networking, file system access and process management drivers. The Linux kernel in Android differs from a regular Linux kernel due to the additional features added to support the components present in the Android stack.

2.5.2 Invasive approaches for memory introspection at different layers of the stack

Modifications made at different stack layers for acquiring app-specific artefacts in memory may incur app or device-invasive approaches. Such invasive approaches can be observed in various works related to malware analysis whereby modifications are made at different stack layers to monitor app or system behaviour. Taint analysis [149, 107] and app monitoring [39] for malware detection may incur an app-invasive approach comprising app repackaging that allows tools to monitor app internal functions and execution flow. Similarly, understanding app internals to map app-specific functionality to hijacked events as attack steps requires an in-depth analysis of the app. For closed-source proprietary apps, this requires app unpacking and compiled code analysis. Such an app-invasive and specific process is operationally infeasible due to the many apps available in app stores. Furthermore, an app-invasive solution may introduce new concerns regarding the stability of the monitored app. In the case of app modification due to static DBI library injection that can enable timely memory dumps, as described in the previous section, app repackaging and re-signing become necessary. This is not only

[7]https://developer.android.com/training/data-storage/sqlite
[8]https://developer.android.com/studio/write/firebase

app-specific and, therefore, operationally infeasible but is also incompatible with stock apps. Moreover, app repackaging is incompatible with system apps as these cannot be uninstalled and may require further compiled code analysis efforts to bypass app-specific anti-repackaging checks.

The lower layers of the technology stack present another possibility for monitoring, including memory introspection, as these layers are shared across apps and, therefore, do not present the operational feasibility concerns as app-invasive and specific approaches. Modifications at lower layers of the stack that could enable taking memory dumps call for modifications of the device, at the Binder layer [109] specifically, the Android runtime [141, 16] or directly modifying the OS (for rooting) [144], to add monitoring functionality not exposed by the default native layers. In the case of memory acquisition, this comprises adding kernel modules [119] that enable memory dumping functionality. Yet such changes require custom devices (having customised kernels) or using a device-invasive approach that requires device rooting, causing reliability and security issues.

Device rooting is an irreversible process unique to the device manufacturers, which weakens the device's default security mechanisms that ensure all apps operate with minimal privileges. Custom devices are less widespread than stock Android devices and may require manufacturer input. Due to limited usage and the multiple Android device manufacturers, such custom devices may be less reliable than stock devices and more challenging to adopt across manufacturers. Furthermore, within the context of incident response, victims of attacks are more likely to own stock devices rather than custom ones. On the other hand, manufacturers can create devices with memory forensic preparedness in mind. This, however, relies on the manufacturer and can only be widely adopted across individual device manufacturers.

The middle layers of the stack present an opportunity for acquiring artefacts from memory using a minimally invasive approach that is neither too app-invasive nor device-invasive, concerning both understanding app internals and memory introspection. Yet, the accuracy of such an approach while aiming to remediate app hijack attack steps, which leave behind app-specific evidence, remains to be explored.

2.6 Android security model and attack surfaces

The Android security model relies on app sandboxing and app isolation, which restricts unauthorised access to system resources and other apps and implements a permission model that allows authorised access to other apps and system resources [11, 121]. The enforcement of application sandboxing is implemented across different layers of the Android technology stack [11]. At the Linux kernel layer, UID-based discretionary access control (DAC) is implemented to assign apps separate UIDs that prevent them from accessing processes and app resources other than their own. SELinux further extends the underlying discretionary access control to allow applications to ask for permissions, and users can grant or deny those permissions. At runtime, each application is executed in a separate runtime that ensures isolation between apps during execution. Processes (apps, services etc.) can communicate with each other through the Binder layer, which enforces security policies between processes based on the permissions granted at the application level [11, 44, 121]. At the application layer, applications request permissions that the application may require during execution, which the user is asked to grant during installation. These permissions are then reflected in permission enforcement implemented at lower layers of the stack [11, 121].

Yet even with these isolation measures, the Android security model exposes attack surfaces that attackers can exploit to compromise an Android device's or its data's security. OS vulnerabilities, including vulnerabilities in the Linux kernel, native libraries and system services, can enable escalation of privilege that grants unauthorised root access and complete control over the device, including malicious interactions with victim apps, which by default is inaccessible to apps [61]. At the Binder layer, vulnerabilities can lead to unauthorised access to other processes, including process data and logic, privilege escalation, or spoofing attacks [114]. Attackers can also target vulnerabilities in individual apps, which affect the context of the targeted app [96]. Moreover, the permission model can be abused through social-engineering attempts that trick users into granting excessive or unnecessary permissions [42, 98, 136]. These attack surfaces expose several possible threats malware developers constantly try to realise through existing attack vectors.

2.7 Summary

The popularity and sensitivity of the apps available on Android devices make Android a popular target amongst attackers. While protection and detection mechanisms are available at the app store and device level, these have limitations. Motivated attackers can employ a series of evasion techniques (ranging from obfuscation to dynamic code loading enabled by Android) that enable them to evade widely-deployed detection techniques.

The ability of malware to evade detection makes incident response necessary. Victims targeted by an attack eventually flag an incident after their consequences become evident. This kickstarts the incident response cycle, which aims to remediate the attack steps, which comprises the use of forensic tools to collect forensic artefacts from the device and generate a forensic timeline that enables the investigator to get a clearer picture of the events on the device. Current tools rely on stored evidence, which may not always be sufficient for complete forensic timeline generation and subsequent identification of malicious events.

Process memory introspection can contribute to the timely collection of app-specific artefacts from memory. However, stock devices do not expose this functionality. Current applications of Android memory introspection have been applied in the context of virtualised sandbox environments, where customised kernels and device rooting are acceptable. Nevertheless, in the context of incident response, such approaches are incompatible with most stock devices and weaken the overall device security (in the case of rooting). Furthermore, state-of-the-art Android memory forensics does not address the timely manner in which evidence of stealthy attack steps from memory must be captured and does not focus on collecting app-specific artefacts from memory that could be hijacked.

Dynamic binary instrumentation is a core enabler for process memory introspection on Android, which allows for memory introspection on stock devices. Yet, collecting app-specific artefacts from memory using existing solutions requires invasive approaches that require modifying the device or app to enable the timely capture of memory dumps. Furthermore, an understanding of app internals is required to map app-specific functionality to hijacked events as attack steps call for an in-depth analysis of the app. This involves app unpacking and compiled code analysis, an app-invasive approach, thus foregoing app protections and involving an app-specific approach that is infeasible to carry out across several apps.

An overview of the Android security model shows the exposed threat landscape that makes room for attack vectors, enabling threat models that allow unauthorised malicious interactions between apps. Stealthy malware following such a threat model and aiming for detection evasion requires an incident response. Due to the limitation of stored forensic evidence, it becomes necessary to look towards memory forensics approaches, which can aid the victim in recovering all the possible artefacts from memory that can reconstruct attack steps. Such an approach must be minimally-invasive, ensuring compatibility with a victim's stock device that is the most likely configuration encountered by forensic investigators without weakening default security or protections. Before delving deeper into the solution, the next chapter details the threat model this work is concerned with and the attack surface it exploits. It describes the various attack vectors that enable it and presents a running example used throughout this thesis.

3 App Hijack Threat Model

The same attack vectors that allow malware to interact maliciously with victim apps can also be used to offload attack steps to them, disassociating the malicious binary from actual malicious logic, thus complicating both code scanning-based detection and incident response. This chapter expands on the App Hijack Threat Model, addressing the first objective ***O1*** of this thesis and making a case for an imminent class of malware that is as hard to detect as much as they could be to respond to. This chapter first presents the threat model (Section 3.1) which the research question of this thesis is concerned with and the attack vectors that enable it (Section 3.2). A motivational case study is presented (Section 3.3), highlighting an app hijack attack involving a messaging app and the current limitations of incident response tools.

3.1 Threat model

The App Hijack threat model exploits vulnerabilities or features that enable interactions between apps, typically used for inter-process communication and which expose app functionality. Malicious apps following the app hijack threat model exploit communication channels between benign apps to hijack benign apps' legitimate functionality and offload attack steps that coincide with the benign app's functionality.

The attacker's capabilities depend on the functionality exposed by the app through inter-process communication. These can range from personal information leakage via reading text messages to financial theft through crypto exchange wallets.

Potential for detection evasion. In essence, the app hijack threat model executes attack steps by hijacking legitimate app functionality through channels that enable interprocess communication. Therefore, any code-based detection attempting to identify malicious logic will be futile since this is never included in the malicious application, whether installed as the original app or dynamically loaded at a later stage. Even if the challenges with EDR implementation on Android had to be surmounted, behavioural detection would still miss malicious interactions with victim apps if that attack vectors are unknown or, even worse, are legitimate.

Potential for reduced forensic footprint. Any retroactive detection focusing on evidence of malicious behaviour is futile if logged attack steps are associated with benign app functionality. Even worse, if the benign app exposes functionality that can erase evidence of previously executed app functionality, the attack can hijack this functionality to erase any traces from logs.

Figure 3.1 shows an example of this threat model in the case of benign messaging app hijack attacks. In this case, attackers are interested in reading incoming messages (*spying*) or sending messages behind the victim's back (*sending* and *deleting* messages immediately). This functionality coincides with the functionality of instant messaging (IM) apps. Yet the initiator of these actions is a malicious actor, and the device owner is unaware of these events.

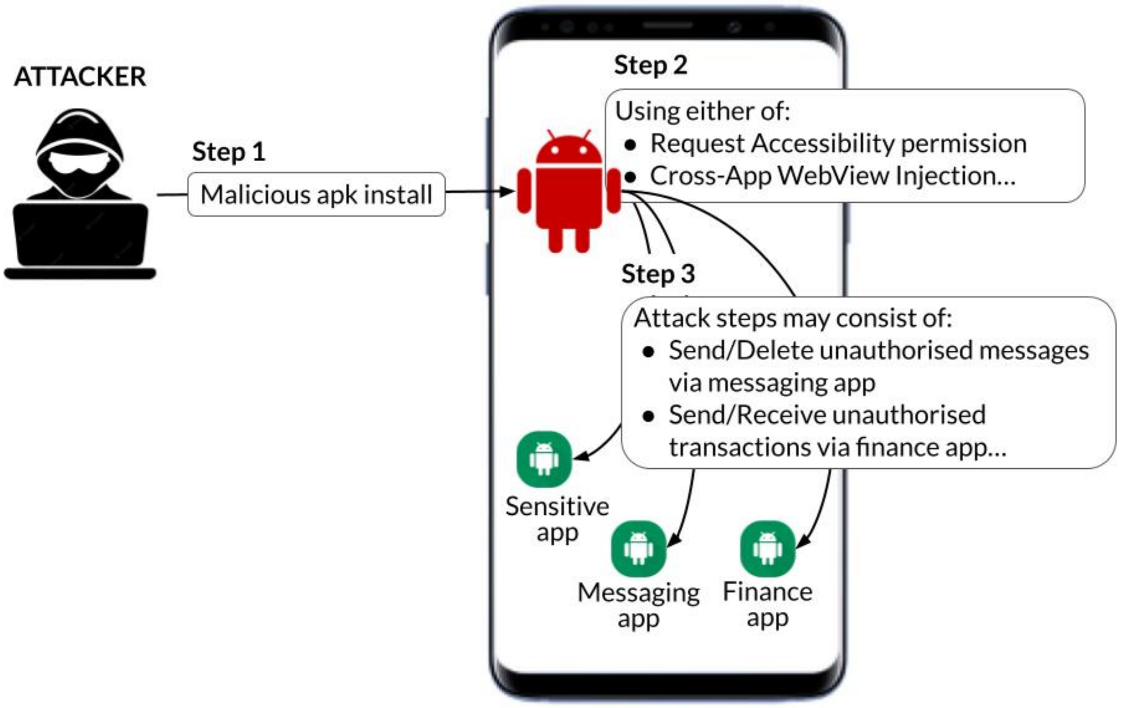

ATTACKER
Step 1
Malicious apk install
Step 2
Using either of:
• Request Accessibility permission
• Cross-App WebView Injection...
Step 3
Attack steps may consist of:
• Send/Delete unauthorised messages via messaging app
• Send/Receive unauthorised transactions via finance app...
Sensitive app
Messaging app
Finance app

Figure 3.1: Malicious apps installed on the victim's device hijack benign app functionality to carry out attack steps and avoid detection.

3.2 Attack vectors enabling app hijack

Overall, any form of inter-app communication, whether for app functionality or testing purposes, can be exploited to enable the app hijack threat model. The Android accessibility attack vector is a case in point. It exploits the accessibility features and functionalities of the Android operating system designed to assist users with disabilities in using their devices effectively, to interact with other apps maliciously. Early instances [58] demonstrated how through phishing and the misuse of accessibility features, a malicious app could steal a victim's credentials and attack targeted benign apps and services by interacting with them without the user's consent. This misuse has since shifted from being leveraged to perform the actual attack to being used to maintain stealth. Eventbot [46] and BlackRock [21] malware only request accessibility permission upon installation; the rest of the permissions required to perform the attack are obtained through the accessibility permission previously granted by the user. Even worse for the victim, the request for accessibility permission can be hidden from the user using UI confusion techniques, such as zero-permission tapjacking [137]. The accessibility attack vector has been shown to enable stealthy Living-Off-the-Land (LOtL) tactics [24], where key attack steps are delegated to benign apps, possibly only requiring the use of malware during an initial setup phase to attain maximum stealth [130]. The limited forensic footprint that they leave behind has also been demonstrated [70]. Listing 3.1 shows how the accessibility attack vector can be leveraged to hijack message-sending and deleting functionality for WhatsApp.

```
// sending message
List<AccessibilityNodeInfo> sendMessage = contentNodeInfo.findAccessibilityNodeInfosByViewId ("
     com.whatsapp:id/send");
AccessibilityNodeInfo sendMessageButton = sendMessage.get(0);
if (!sendMessageButton.isClickable()) {
    return;
}
sendMessageButton.performAction (AccessibilityNodeInfo.ACTION_CLICK);

// deleting sent message
List<AccessibilityNodeInfo> lastMessage = contentNodeInfo.findAccessibilityNodeInfosByViewId ("
     com.whatsapp:id/message_text");
lastMessage.get(lastMessage.size() -1);
lastMessage.performAction (AccessibilityNodeInfo.ACTION_CLICK);

List<AccessibilityNodeInfo> deleteMessageButton = contentNodeInfo.
     findAccessibilityNodeInfosByText ("Delete");
deleteMessageButton.performAction (AccessibilityNodeInfo.ACTION_CLICK);

List<AccessibilityNodeInfo> deleteForMeOnlyMessageButton = contentNodeInfo.
     findAccessibilityNodeInfosByText ("Delete for me");
deleteForMeOnlyMessageButton.performAction (AccessibilityNodeInfo.ACTION_CLICK);
```

Listing 3.1: Use of accessibility to send and delete a WhatsApp messages without the user's knowledge.

Cross-App WebView Infections (XAWI) [71] exploits the WebView component exposed in mobile applications, exposing the security risks of navigating an app's WebView through a URL. While a legitimate need for displaying the app's UI exists to enable cross-app interactions, its abuse can lead to cross-app remote infection when misused by malware that maliciously interacts with benign apps. In the case of messaging, attackers can misuse this functionality to proxy messages via another benign app that exposes this functionality, as shown in Listing 3.2. While XAWI refers to an attack vector as part of legitimate app functionality, other unprotected app components exposed through a benign application's attack surface can also serve as potential attack vectors.

```
fb-messenger-secure://autocompose/post?id=userid&ttyype=2&s=1&m=content
```

Listing 3.2: URL scheme used to send a Facebook message without the user's consent [71].

Another example vector is SMASHeD [81], which exploits the Android debug bridge. It enables malicious apps, requiring developer options to be enabled and requesting only the `INTERNET` permission, to read and write to multiple sensor data files at will, thus circumventing the Android sensor security model to stealthily sniff as well as manipulate many of the Android's restricted sensors (even touch input). PHYjacking [137] goes a step further and demonstrates how physical inputs used for authorisation methods (e.g., fingerprint scanning), can also be hijacked through a threat model that exploits Android app implementation flaws found in 44% of 3000+ apps tested as well as a powerful race-condition attack that can break the Android Activity lifecycle model. Crucially, the threat model presented requires zero permissions, thus minimising the malware component and bypassing permission-based detection mechanisms. Zygote and binder infection combined with a rooting exploit [127] and third-party library infections [42] provide further attack vectors.

3.3 Motivational case study

WhatsApp Pink [7] is the closest possible malware to the threat model. It is an Android messaging worm that hijacks pre-installed benign versions of instant messaging apps, including WhatsApp. It propagates between contacts by hijacking legitimate pre-installed instant messaging apps' functionality, leveraging the XAWI attack vector to send messages via specific apps for which the URL scheme is known. The following sections describe an example incident scenario, the modifications that can be made to WhatsApp Pink to make it even stealthier and the limitations of state-of-the-art incident response tools in detecting and remediating the attack.

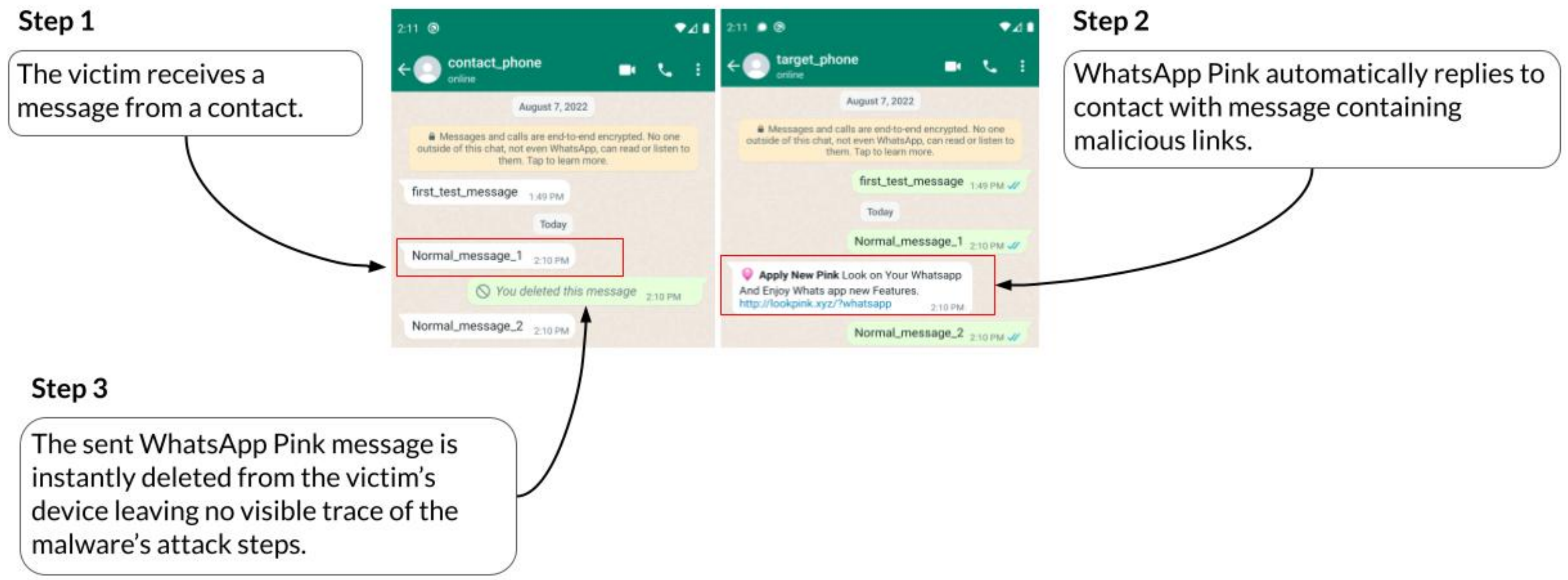
3.3.1 Incident Scenario

The scenarios involves a victim Android device owner who is an employee at their workplace and has a legitimate version of WhatsApp app installed on their Android device. They have been at the receiving end of a social engineering phishing campaign and have unknowingly installed the WhatsApp Pink malware on their Android device. The malware propagates by automatically replying to incoming messages with a download link to the malware itself, akin to a message proxying attack.

WhatsApp Pink. Once this WhatsApp Pink is downloaded and installed on the victim's device, it requests some permissions. Namely, it requests for the *Notification Access* permission, which in conjunction with Android's Direct Reply action, is used by the malware to achieve wormability by responding to incoming WhatsApp messages with a custom message. The app also requests permissions to draw over other apps and to ignore battery optimisation, which allows it to run in the background and prevents the system from killing it off for any reason.

Figure 3.2 illustrates how WhatsApp Pink operates. The malware runs in the background and waits for a legitimate WhatsApp notification. When this happens, the malware auto-replies to the victim's contacts using the custom URL scheme provided by the official WhatsApp app with a message containing a malicious link. The malware propagates via legitimate WhatsApp messages to contacts that send a message to the victim. Upon the receipt of a message, the malware automatically replies to the sender with a link that installs the malicious app, provided that the last message received by the device owner (victim of malware) was sent more than an hour before to avoid raising suspicion among the victim's contacts and maintain stealth.

The aim of this particular malware seems mainly to be used in an adware or subscription scam campaign; however, it could be used for much worse. It could distribute more dangerous threats (banking trojans, ransomware, or spyware) since the message text and link to the malicious app are received from the attacker's server [7].



Step 1

The victim receives a message from a contact.

2:11

contact_phone
online

August 7, 2022

Messages and calls are end-to-end encrypted. No one outside of this chat, not even WhatsApp, can read or listen to them. Tap to learn more.

first_test_message 1:49 PM

Today

Normal_message_1 2:10 PM

You deleted this message 2:10 PM

Normal_message_2 2:10 PM

2:11

target_phone
online

August 7, 2022

Messages and calls are end-to-end encrypted. No one outside of this chat, not even WhatsApp, can read or listen to them. Tap to learn more.

first_test_message 1:49 PM

Today

Normal_message_1 2:10 PM

Apply New Pink Look on Your Whatsapp And Enjoy Whats app new Features.
http://lookpink.xyz/?whatsapp 2:10 PM

Normal_message_2 2:10 PM

Step 2

WhatsApp Pink automatically replies to contact with message containing malicious links.

Step 3

The sent WhatsApp Pink message is instantly deleted from the victim's device leaving no visible trace of the malware's attack steps.

Figure 3.2: Enhanced WhatsApp Pink attack steps. The device on the left is the target device running the malware and the device on the right is the contact device from which a message was sent to the target.

Table 3.1: Digital investigation configurations.

Incident Response step	**XRY**	**Oxygen Forensics**	**Belkasoft**	**AXIOM**
Collection	XRY Agent	OxyAgent	Belkasoft Agent	Magnet AXIOM Agent
Parsing	XRY XAMN	Oxygen Detective	Belkasoft	Magnet AXIOM
Analysis	Timesketch			

Evolving WhatsApp Pink to reduce its forensic footprint. As is, the WhatsApp Pink malware evades detection since the call to various URL schemes is acceptable application logic and, therefore, is not detected as a possible app hijack attack. However, application logs may show this activity (depending on the application development), thus rendering the malware less stealthy. Further improvements can be made to the malware to reduce its forensic footprint, causing full attack step reconstruction during incident response impossible.

Most messaging apps expose the functionality to delete sent messages. This functionality, however, is not exposed over a URL scheme but rather from the app's UI. In this case, the accessibility attack is a suitable attack vector that allows the malicious app to hijack the benign messaging app's delete functionality to hide its attack steps (as shown in Figure 3.2) through the code shown in Listing 3.1 (*lines 14-18*).

3.3.2 State-of-the-art Mobile Forensics tools

Responding to an incident relies on three main steps: i) Collection, ii) Parsing, and iii) Analysis of evidence to produce a timeline of events. Existing forensic tools are typically equipped with collection and parsing features, enabling an incident responder to analyse the forensic timeline produced through the available tools. Using forensic tools: Belkasoft Evidence Centre X,[1] MSAB's XRY,[2] Oxygen Forensics[3] and AXIOM,[4] an attempt is made to recover key attack steps of the app hijack attack, using the set up as shown in Table 3.1 and a physical Android device. The table describes the sources gathered during the collection, parsing and forensic timelining tools used for analysis.

[1]`https://belkasoft.com/android`
[2]`https://www.msab.com/`
[3]`https://oxygenforensics.com/en/`
[4]`https://www.magnetforensics.com/products/magnet-axiom/`

Forensic tools setup. The forensic tools were set up to perform agent-based logical collection, which comprised collecting all files available on the device's file system through the installation of respective forensic tools' apps called *agents*. Since a physical phone was used, any collection step that required device rooting was skipped to avoid irreversibly weakened device security. In addition to the files collected by the tools, the following sources were also provided to the tools: i) app data collected from the app's internal `/data/data` folder, obtained by setting the target app's `debuggable=true` property set in the `AndroidManifest.xml`, ii) data gathered from an `BackupAgent` output, and iii) *logcat* and *dumpsys* logs.

3.3.3 Results from Incident Response tools

Figure 3.3 shows the ground truth timeline of known events executed by WhatsApp Pink, via the hijacked benign WhatsApp app. Figures 3.4 to 3.6 show the forensic timelines generated when using the digital forensic tools XRY, Oxygen, Belkasoft and AXIOM, respectively, set up as described in Table 3.1. While none of the forensic tools' outputs explicitly indicated that a WhatsApp message was stealthily deleted, some events indicating possible attack steps were still recovered by some tools. XRY's output does not show the message-sending event at all. Since the content was missing in the WhatsApp databases (due to deletion), the event was not even displayed to the investigator. Using Oxygen, a message send event is shown, yet there is no information about the message sent. Belkasoft and AXIOM retrieved the event of a WhatsApp Pink message being sent. However, critical metadata that could be used for event correlation and that provides more context, e.g. the message content, was missing.

Without the message content, it is unclear whether this was: a simple message deleted by the target victim, a message with no content, or a malicious message propagated by the malicious app. While an event comprising an empty message is suspicious, timeline generation requires event correlation, which uses metadata to combine attack steps from different sources. Furthermore, remediating attack steps as part of incident response means knowing also the metadata of such events so that their affect can be mitigated.

3.4 Summary

The primary focus of the research question presented in this thesis is stealthy attacks that aim to go undetected for longer by hijacking legitimate benign app functionality of targeted benign apps to perform attack steps. The victim is a general Android user with a stock Android device lured into installing a stealthy malware app that bypasses

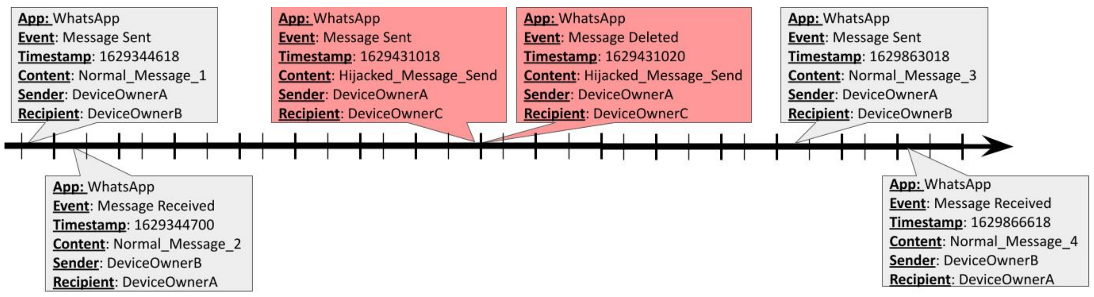


Figure 3.3: WhatsApp Pink incident ground truth forensic timeline.

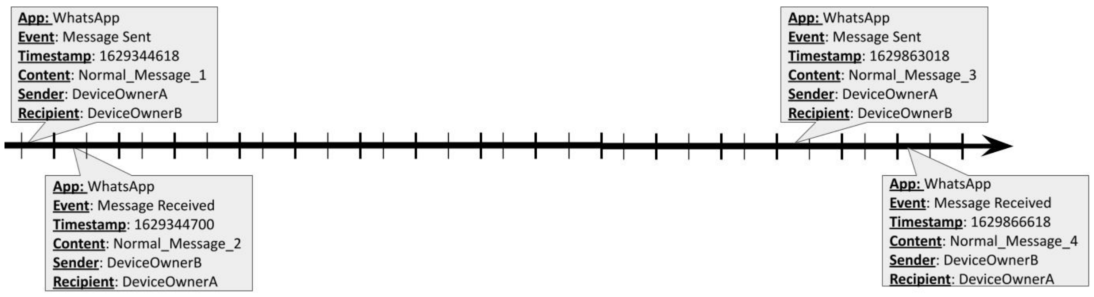


Figure 3.4: WhatsApp Pink incident forensic timeline obtained using XRY.

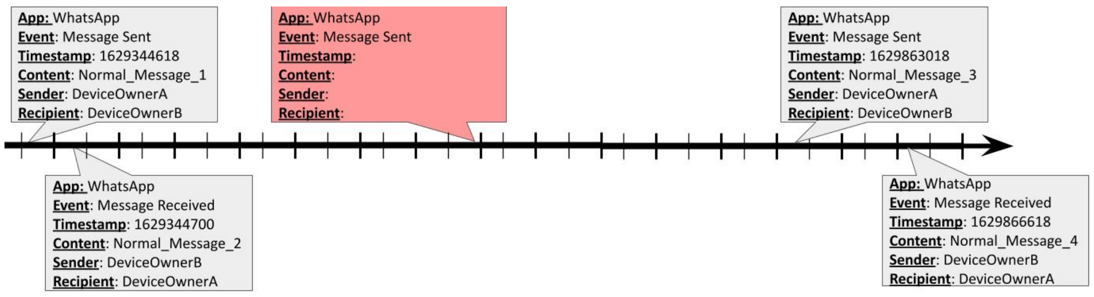


Figure 3.5: WhatsApp Pink incident forensic timeline obtained using Oxygen Forensics.

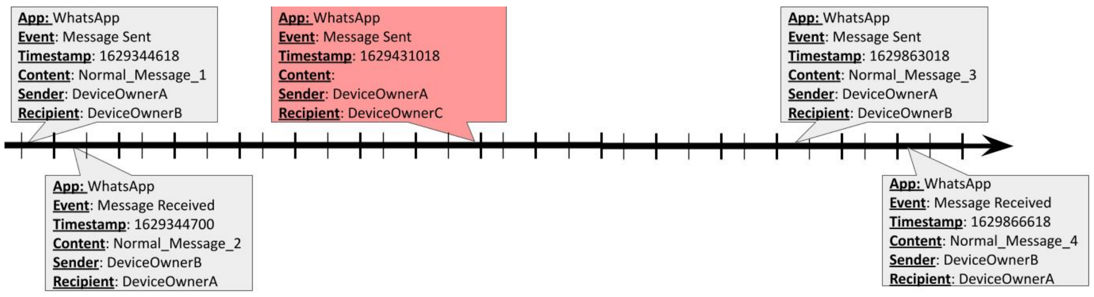


Figure 3.6: WhatsApp Pink incident forensic timeline obtained using Belkasoft and AXIOM (the same timeline was obtained using both tools).

all existing protections and detection mechanisms in the Google Play Protect suite. The malware leverages attack vectors typically found on stock Android devices and apps to enable inter-app communication with sensitive apps, which the attacker can hijack to carry out attack steps.

The example incident scenario presented in this chapter shows that existing stealthy attack vectors can significantly increase the stealth of existing malware by offloading attack steps to benign apps and reducing their forensic footprint. By hijacking benign app functionality that coincides with the malware attack steps (that is, worm propagation through message sending), this threat model enables the attack to go undetected. Furthermore, any efforts of remediation using existing state-of-the-art Android forensic tools are futile due to their reliance on third-party application logs which may be incomplete or erased by malware, and which leave investigators with an incomplete forensic timeline of the attack steps. Since malware must execute in memory regardless of its steps, any possible evidence of attack steps may be found in timely-collected memory dumps. Throughout the rest of the thesis, this incident scenario setup will be used as a benchmark and example, with which the main proposal of this thesis is explained and evaluated.

4 Just-in-Time Memory Forensics

The previous chapter showed how existing forensic sources fall short of identifying key attack steps of app hijack attacks, that the research question in this thesis aims to address, within the context of messaging. Previous works (Chapter 2) have identified memory as a potential source of information that can contain evidence not available in stored evidence and not fully leveraged by existing forensic tools, to remediate attack steps as part of incident response.

This chapter describes Just-in-Time Memory Forensics (JIT-MF), a proposed framework that aims to address the challenges of timely acquisition of app-specific artefacts from Android memory to respond to app hijack threats. Timely acquisition of artefacts in memory refers to an approach that ensures acquisition occurs while any relevant evidence is still in memory, rather than attempting acquisition too late when the evidence in memory is lost due to the ephemeral nature of volatile memory. By proposing a framework for the timely collection of app-specific artefacts from memory, this chapter presents the first step towards the second objective of this thesis ***O2***, as it provides a means to explore the hypothesis set in this thesis. The main challenges and requirements of JIT-MF are outlined (Section 4.1). Subsequently, an overview of the framework is presented (Section 4.2), along with its main components and how these provide a means to address the challenges and requirements described. A prototype implementation is outlined (Section 4.3) and the assumed operational context of JIT-MF is described along with how it is incorporated within the incident response cycle (Section 4.4). Finally, a security analysis of JIT-MF (Section 4.5) assesses the attack surfaces that JIT-MF exposes and how these can be remediated using existing mitigation and protection measures.

4.1 Requirements for timely-acquisition of app-specific artefacts from memory

The second objective (***O2***) of this thesis is to understand how to perform the timely collection of app-specific artefacts from memory that contributes to app hijack attack steps in a minimally invasive manner; that is, by meeting the following requirements:

- *Compatible with stock devices* - In an incident response setting, stock Android devices as shipped by the manufacturer, are the most likely device configuration encountered by forensic investigators [133]. While custom devices can be used in specifically designed scenarios, this solution aims to target *any* stock device on which an incident occurs. Thus, aiming to provide a solution regardless of the device, ensuring applicability across different devices.
- *Minimally invasive at the app and device levels* - Invasive approaches weaken device and app security. Furthermore, they pose operational and stability concerns due to their app and device-specificity (Chapter 2, Section 2.5). Therefore, a minimally invasive approach is required to ensure that: i) the device and app's security mechanisms are retained after incident response when the device owner continues using it; and ii) JIT-MF is compatible across apps and devices with minimal operational effort.

Android does not expose the functionality to collect evidence from memory. Therefore, any solution that aims to work with any stock Android device and is minimally invasive seems to contradict the objective of gathering app-specific evidence from memory required to respond to an app hijack attack.

4.2 JIT-MF framework overview

Just-in-Time Memory Forensics (JIT-MF) is the framework proposed to demonstrate the thesis that the timely and non-invasive logging of hijacked functionality can be made possible through process memory introspection, allowing for full attack steps recovery. Unlike malware detection and forensics tools, JIT-MF focuses on collecting evidence from hijacked benign target apps (rather than malware) that become the targets of app hijack attacks.

The main components of JIT-MF are: i) *Evidence_object*, the app-specific objects whose presence in memory implies the execution of some specific app functionality, possibly a hijacked attack step; ii) *Trigger_point*, the instructions associated with the presence of *Evidence_object*s in memory which invoke a memory dump; and iii) *JIT-MF_Logs* which is the output comprising *Evidence_object*s produced from the triggered memory dump (via *Trigger_point*). *Trigger_point*s and *Evidence_object*s are defined within JIT-MF Drivers (by JIT-MF Driver developers) that drive the evidence collection process from memory for a specific app hijack scenario. These two components are selected based on specific app functionality of interest. The installation of JIT-MF with JIT-MF Drivers produces *JIT-MF_Logs*. These components are formally defined below.

Definition 4.2.1. Let an observed process P be represented as a finite list of tuples, where i_j represents an instruction that is executed in the process and s_j represents the process state at instruction i_j. The process states s_j contain all the objects in volatile memory during the process runtime.

$$P = \{(i_1, s_1),\ (i_2, s_2),\ \ldots,\ (i_j, s_j)\}$$

Definition 4.2.2. App functionality of interest AF can be defined as follows, where each AF_j is app-attack specific, embodying an instance of an attack step offloaded to hijacked app functionality.

$$AF = \{AF_1,\ AF_2,\ \ldots,\ AF_n\}$$
$$where\ AF_j \subset P$$

Definition 4.2.3. Given the definitions above, *Evidence_object*s EO, associated with a specific app functionality of interest (AF_j), can be represented as the result of a function *Object_collector* oc. Given a process state s_j, the *Object_collector* function carves a set of specific objects at *location* with *size* from memory. The definition below considers non-contiguous data structures.

$$EO_{AF_j} = \{(object\_location_1,\ object\_size_1),\ (object\_location_2, object\_size_2),$$
$$\ldots, (object\_location_n, object\_size_n)\}$$
$$oc_{AF_j}\ =\ s_j \mapsto\ EO_{AF_j}$$

Definition 4.2.4. *Trigger_point* tp, associated with a specific app functionality of interest (AF_j), can be represented as the following function that, given a process instruction i returns a *verdict* of type $true, false$, which signifies whether objects in memory should be dumped ($true$), or not ($false$).

$$tp_{AF_j} = i_j \mapsto verdict$$

The function tp can use predicates, possibly related to the device state, that influence the verdict output.

Definition 4.2.5. Given the definitions for *Trigger_point*s and *Evidence_object*s, JIT-MF Drivers D_P for a process P can be defined as a set of tuples as shown below. D_P follows the set of AF_j, with each (oc_{AFj}, tp_{AFj}) being app-attack specific.

$$D_P = \{(oc_{AF_1}, tp_{AF_1}), (oc_{AF_2}, tp_{AF_2}), \ldots, (oc_{AF_n}, tp_{AF_n})\}$$

Definition 4.2.6. *JIT-MF_Logs* are defined as the list of timestamp and *Evidence_object EO* tuples returned by *JIT-MF*.

$$\textit{JIT-MF\_Logs} = [(@timestamp_1, EO_1), (@timestamp_2, EO_2), \ldots, (@timestamp_n, EO_n)]$$

Definition 4.2.7. Given these definitions, JIT-MF can be defined as a function that, given a process P and the driver for that process D_P and a universal clock C, produces *JIT-MF_Logs*.

$$\textit{jit-mf} \ = \ P \times D_P \times C \mapsto \textit{JIT-MF\_Logs}$$

The pseudo-code below (Algorithm 1) describes the *JIT-MF* function internals that produces *JIT-MF_Logs*. For every instruction and state tuple in process P, every JIT-MF Driver D_P associated with the process P, comprising tps and ocs associated with specific app functionality of interest (AF_j), if tp's *verdict* is *true*, *JIT-MF_Logs* are populated with a tuple comprising the current $timestamp$ and *EO*s in the current state related to app functionality of interest.

Algorithm 1: JIT-MF function (Definition 4.2.7) implementation

```
Input  : Process $P$, Driver $D_P$, Clock $C$
Output : JIT-MF_Logs=[]
function jitmf(Process P, Driver $D_P$, Clock C):
    foreach $(i_j, s_j) \in P$ do
        foreach $(oc_{AF_k}, tp_{AF_k}) \in D_P$ do
            if $tp_{AF_k}(i_j)$ then
                $JIT\text{-}MF\_Logs.append(Clock.getTime(), oc_{AF_k}(s_j))$
            end
        end
    end
    return JIT-MF_Logs
```

4.2.1 Evidence object and trigger point selection

The evidence collected using JIT-MF relies heavily on the selection of *Evidence_object*s and *Trigger_point*s, which rely on the selected app functionality of interest. Yet the selection of app functionality of interest, and the *Evidence_object*s and *Trigger_point*s that depend on it is not prescriptive. An app may have multiple functions; however, not all these functions can be considered app functionality of interest. Only some may be of interest to attackers for potential hijacking, depending on the aim of the app hijack attack. For instance, in the case of a messaging hijack attack, app functionality of interest would comprise message sending or reading functionality that an app hijack attack could misuse for proxying or stealing messages.

It follows that, not all objects in memory upon the execution of app functionality of interest may be valid *Evidence_object*s. Only some may correspond to evidence of possibly hijacked app steps, which are the critical objects enabling the app functionality of interest. For instance, in the case of a messaging hijack attack, the in-memory objects of interest are those supporting the execution of messaging functionality and which may be hijacked during eventual attacks. *Evidence_object*, in this case, comprise precisely those objects in memory that contain the messages themselves.

Trigger_point functions define when objects in memory should be dumped, based on the current process instruction and some predicate (Definition 4.2.4). This predicate should be selected so that the *Trigger_point* function returns true when the current process instruction implies that *Evidence_object*s are in memory. While several instructions may lead to the presence of *Evidence_object*s in memory, the following operations on data objects can serve as a guide for JIT-MF Driver developers to select relevant instructions as *Trigger_point*s.

1. **Storing** and **loading** from storage;
2. **Transferring** over the network (e.g. Wi-Fi, 4G, etc.); or else
3. **Transforming** in some way (e.g. display on screen etc.).

In the case of an app hijack attack involving a messaging app, malware aims to proxy messages through a benign messaging app. The messaging app runs in a process $P = \{(call\ open(), Memory_1),\ (call\ send(), Memory_2),\ (call\ receive(), Memory_3),$ $(call\ exit(), Memory_4)...\}$. The app functionality of interest here comprises message sending functionality that can be hijacked by malware following the app hijack attack threat model and targeting the messaging app; that is, $AF = \{(call\ send(), Memory_2)\}$. In this case, the *Evidence_object* selected is the *Message* object (as defined by an app-specific class), where $oc(Memory_2) = EO = \{(addressof(Message_1), sizeof(Message_1))...\}$. The operations related to these objects involve storing/loading messages from local content repositories and sending/receiving messages over communication networks. Therefore, a network `send` instruction can be selected as a *Trigger_point*; that is, $tp(call\ send()) = true$. While set individually, the combined definition of *Trigger_point* and *Evidence_object* can give context into the hijacked app functionality. Therefore, JIT-MF having a JIT-MF Driver $D_P = \{(oc, tp)\}$ with a message object set as *Evidence_object* and *send()* as *Trigger_point* results in JIT-MF_Logs whose contents are indicative of app functionality that involves message sending *jit-mf* = *JIT-MF_Logs* $= [(@timestamp_1, Message_1)...(@timestamp_n, Message_n)]$. For example, in the case of the WhatsApp Pink worm attack steps shown in Figure 3.2, *JIT-MF_Logs* $= [...(1659880140, Message\{``content" : ``first\_test\_message", ``in" : True, ``time" : ``1659880140", ``contact" : ``contact\_phone"\}), (1659881400, Message\{``content" : ``Normal\_message\_1", ``in" : True, ``time" : ``1659881400", ``contact" : ``contact\_phone"\}), (1659881400, Message\{``content" : ``Apply\ New\ Pink\ Look\ on\ Your\ ...", ``in" : False, ``time" : ``1659881400", ``contact" : ``contact\_phone"\}), (1659881400, Message\{``content" : ``first\_test\_message", ``in" : True, ``time" : ``1659881400", ``contact" : ``contact\_phone"\})\ ...]$.

4.2.2 JIT-MF within the Android technology stack

JIT-MF installation can be implemented at any layer within the Android technology stack. Similarly, *Evidence_object*s and *Trigger_point*s selection can comprise objects and instructions at different stack layers, respectively. Yet, the different layers carry different implications related to app and device invasiveness which may not align with the requirements set at the beginning of this chapter. Table 4.1 presents the different stack layers along with the type of *Evidence_object* and *Trigger_point* that can be selected at each layer. Every stack layer also has two types of implications (separated by '/' in the table) that are of concern: i) the installation of JIT-MF and ii) the comprehension of layer internals required for *Evidence_object* and *Trigger_point* selection.

Regarding JIT-MF installation, the lower three stack layers (Binder - OS, and Android or Java APIs whose libraries are prepackaged on the device) require device rooting since modifications need to be made to the underlying kernel and OS to make JIT-MF operational in these layers. However, minimal device invasiveness is a requirement of JIT-MF, making these three layers unsuitable for JIT-MF installation. Simultaneously, modifications for JIT-MF installation in the upper layers require app repackaging.

App repackaging presents an opportunity that does not outright invalidate the security of the whole device. Yet it does not comply with the requirements set out as it calls for an app-invasive approach that is incompatible with app default protections

Table 4.1: *Evidence_object*s and *Trigger_point*s available per layer, along with the Implication of installation / Implication of internals comprehension, per layer.

Stack Layer	Evidence Object	Trigger Point	Implications
App	App data objects	App functions	App repackaging / App-specific (typically closed-source) code
API	App data embedded in API objects	API functions	App repackaging (for third-party APIs). Custom kernel - device rooting and reflashing (for Android and Java APIs) / Well-documented API interface with app-specific usage
Binder	Android data objects	Android internal functions and runtime	Custom kernel - device rooting and reflashing / Well-documented Android Binder internals
Native	Native objects (C primitives)	Native functions	Custom kernel - device rooting and reflashing / Well-documented native runtime internals
OS	Kernel objects	Kernel instructions	Custom kernel - device rooting and reflashing / Well-documented Linux and Android-specific internals

that perform app-specific integrity checks. App-level virtualisation, however, offers a possible remediation whereby JIT-MF can be installed at the app level within the app runtime environment without requiring app repackaging. JIT-MF installation at the API layer is marginally more cumbersome than installation at the app layer as it requires modification of the API within the app where it is in use. App repackaging is still in the end required to repackage the modified API. Therefore, JIT-MF installation at the app-level seems to be the better trade-off overall.

Regarding *Evidence_object* and *Trigger_point* selection, the lower layers of the stack (Binder - OS) may be much better documented, whether open or closed-sourced. The opposite applies to the app layer, where the code is typically closed-source, optimised and obfuscated. The API layer is the middle-ground, whereby the interface of the API may be well-documented, but their usage within the app may involve app-specific aspects. Therefore increased comprehension efforts of the respective layer internals is required in the topmost layers of the stack, even more so at the app layer. The increased comprehension effort required for the stack's topmost layer calls for an app-invasive approach, comprising an in-depth analysis of the app that requires app-unpacking and compiled code analysis that does not comply with the requirements set out for JIT-MF. Furthermore, such an approach would be infeasible to adopt across multiple apps due to the app-specific effort required to comprehend individual app logic.

The resulting *JIT-MF_Logs* produced comprising *Evidence_object*s from the app layer are most likely to include the relevant evidence to remediate app hijack attacks that leave evidence of attack steps through app artefact in memory. *Evidence_object*s at lower levels in the stack comprise objects that are not app-specific and do not align with the exact app artefact involved in the hijacked app step. Thus, this evidence will contribute to a less accurate forensic timeline. Similarly, *Trigger_point*s at lower levels are more generic and less tightly coupled app-specific events that result in possibly hijacked app artefacts in memory.

Therefore, while selecting *Evidence_object* and *Trigger_point* at higher levels in the stack results in a more accurate forensic timeline, this results in an app-invasive approach that is infeasible across apps. Simultaneously, selecting *Evidence_object* and *Trigger_point* at lower levels in the stack is more feasible due to the shared logic across apps. Yet, the selection from this layer results in a less accurate forensic timeline.

4.3 JIT-MF framework implementation

JIT-MF Drivers drive the evidence collection process from memory, while JIT-MF Driver Runtime provides the services required by the JIT-MF Drivers. JIT-MF Drivers are specific to the *Evidence_object*s and *Trigger_point*s set, yet the JIT-MF Driver Runtime is common across different JIT-MF Drivers.

Figure 4.1 demonstrates how the JIT-MF framework operates. JIT-MF Drivers first filter an incoming trace of instructions based on some *Trigger_point* and only proceed to the next step if *Trigger_point* returns *true*. The JIT-MF Driver then searches for relevant *Evidence_object*s in memory and generates output (step 3) by carving *Evidence_object*s from memory, including any metadata of the evidence object. The result is appended to *JIT-MF_Logs*, which may need to be parsed to reveal app-specific artefacts. The services required by the JIT-MF Driver to register *Trigger_point* (that is, place hooks on instructions), collect and carve *Evidence_object* from memory, and store the evidence to *JIT-MF_Logs* are provided by the JIT-MF Driver Runtime. A tool that implements and automates the JIT-MF elements shown in Figure 4.1 is a JIT-MF tool.

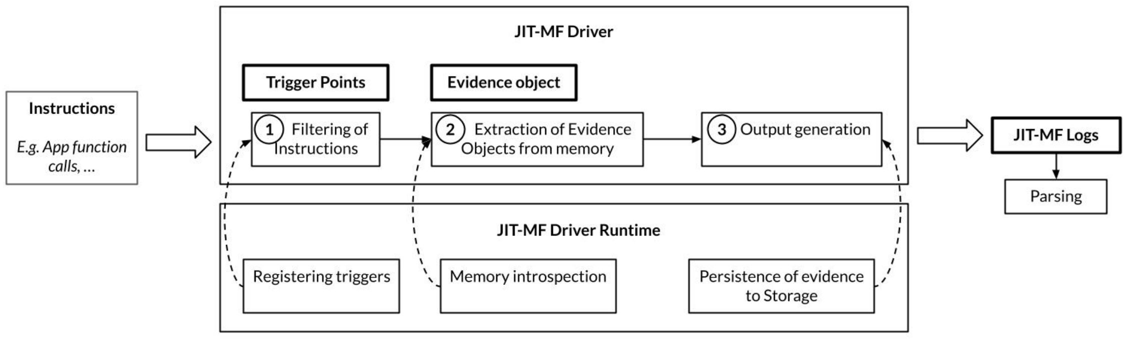


Figure 4.1: JIT-MF Concept.

4.3.1 Offline and online collection

Evidence_object collection from memory through function *oc* (Definition 4.2.3) can be defined as online or offline. That is, the function oc_{AF_j} can either carve EO_{AF_j} from memory in an *online* fashion, as soon as the *tp* returns *true* (Algorithm 1 *line 4*. This frugal approach leverages ART's Garbage Collector (GC) to dump the specific *Evidence_object* in memory at runtime. Otherwise, the *JIT-MF_Logs* can be populated with a memory dump comprising entire ART heap sections as in `hprof` dumps, with oc_{AF_j} carving *Evidence_object*s *offline* using an `hprof` parser, e.g. Eclipse MAT.

Table 4.2: A summary of the object collection options, given the online and offline carving and parsing approaches.

	Offline parsing	Online parsing
Offline carving	Offline collection	-
Online carving	Online collection	Online collection

Object carving refers to identifying and extracting in-memory objects from live memory (in the online case) or a memory dump (in the offline case). Object parsing follows from object carving, as it involves extracting meaningful information from the carved raw object bytes, e.g. the timestamp of a messaging event, its contents, etc. This operation is based on a class definition obtained through code comprehension or publicly-available documentation in the case of open-source code. Table 4.2 shows how offline evidence collection results from the object carving and parsing stages carried out offline; that is, at a later stage on a dumped partial memory image. In the case of online collection, object carving is carried out online as specific objects are carved out from live process memory and then dumped; however, object parsing can still be carried out online or offline. Objects may be parsed in real-time or later (offline), depending on the computational effort required to parse the object and retrieve the meaningful information.

4.3.2 JIT-MF Driver and Runtime

JIT-MF tools built on the JIT-MF framework must implement or leverage existing tools to create a JIT-MF Driver and Runtime. Listing 4.1 and Listing 4.2 present templates that generically describe the make-up and functionality of JIT-MF Driver and Driver Runtime, respectively.

Lines 1-3 identify the driver (*Driver_ID*) and link it with its intended app/incident *Scope* as well as the *Processes* that are of interest in the case of apps having multiple

```
1  Driver_ID: string
2  Scope: <app, incident_scenario>
3  Processes:[optional <namespace.classname>,...]
4  /* Attributes */
5  Evidence_objects: {<event: string,object_name: string,carve_object_type(),
       parse_object_type(),{trigger_ids}>, ...}
6  Trigger_points: {<trigger_id: string,<hooked_function_name: string,level: {native|rt},
       trigger_predicate(), trigger_callback()>>,....}
7  Collection_method: {online | offline}
8  Parsing_method: {online | offline}
9  Log_location: string
10 Globals: optional {<key,value>, ...}
11
12 /* Exposed interface */
13 bool init (config: {<key,value>, ...}) {
14   for entry in Trigger_points:
15     place_native_hook() ⊕ place_rt_hook();
16 }
17 /* Internal functions */
18 bool trigger_predicate$_i$(params: {<key,value>, ...}) {
19   decide on whether to fire the corresponding trigger;
20 }
21 void trigger_callback$_i$(thread_context: {<key,value>, ...}) {
22   if trigger_predicate$_i$() :
23     perform memory forensics on the current app state;
24 }
25 [object: address,...] carve_object_type$_j$(from: address, to: address) {
26   if Collection_method == online:
27     attempt object carving in the given memory range;
28   else:
29     carve_object_type$_j$_offline(from, to);
30 }
31 @OFFLINE
32 [object: address,...] carve_object_type$_j$_offline(from: address, to: address) {
33   use an hprof parser to carve object$_j$ in the given memory range;
34 }
35 [<field:value>,...] parse_object_type$_j$(at: address) {
36   if Parsing_method == online:
37     parse object fields starting at the given address;
38   else:
39     parse_object_type$_j$_offline(at);
40 }
41 @OFFLINE
42 [<field:value>,...] parse_object_type$_j$_offline(at: offset) {
43   if Collection_method == online:
44     use custom parser to parse object$_j$ at the given offset
45   else
46     use an hprof parser to parse object$_j$ at the given offset from memory dump;
47 }
```

Listing 4.1: JIT-MF Driver template.

processes; that is, those classes in the app that fire the selected trigger points but are run as a separate process. If the *Processes* attribute is not set, the app is assumed to have only one main process, initiated by the class containing the main activity.[1] *Lines 5-10* enlist a driver's attributes and their types (*attribute* : *type*), with tuples denoted by $<>$, sets by $\{x, y, z...\}$, ordered lists by [], key-value pairs by $< key, value >$ and enumerations

[1]https://developer.android.com/guide/components/activities/intro-activities

with $\{val1|val2|...\}$. Function parameters are identified by the final parenthesis $()$, which correspond to the drivers' internal functions (*lines 21-50*). *carve_object_type*$_j$*_offline* and *parse_object_type*$_j$*_offline* (*lines 32-34* and *42-46* respectively), annotated with `@OFFLINE`, are the offline counterparts of *carve_object_type*$_j$ and *parse_object_type*$_j$ respectively, executed *after* a memory dump is taken. *Globals* is a key-value meant for miscellaneous usage; for instance, a variable `timestamp` is set by one function and used by another and is made accessible via the *Globals* variable.

Lines 5, 6 define the main elements of the specific JIT-MF Driver (as described in Definition 4.2.5), through the definition of the *Evidence_object* and *Trigger_point*, respectively. *init*$()$ presents the only interfaces exposed to the JIT-MF tool's main environment. It is called during tool initialisation and sets up registers *Trigger_point* hooks by calling *place_native*$|$*rt_hook*$()$. This function returns a boolean (*bool*) indicating success or otherwise. *Trigger_predicate*$()$ and *Trigger_callback*$()$ must be defined per entry in *Trigger_point*s, whereby *Trigger_callback*$()$ is executed if the hooked *Trigger_point* instruction executes and *Trigger_predicate*$()$ is True (as shown in Algorithm 1 *line 4*). *Trigger_point*s may concern either *native* or *rt* function hook, with the latter implying the device's runtime environment, e.g. ART in the case for Android. The same applies for *carve_object_type*$()$ and *parse_object_type*$()$, which have to be defined per entry in *Evidence_objects*, at least for online *Collection*. In scenarios where online collection is opted for and additional supporting objects need to be carved and parsed to obtain the necessary metadata (e.g. recipient object containing the recipient's metadata), supplementary *carve_object_type*$()$ and *parse_object_type*$()$ need to be created for the said object, whereas the *Evidence_object* remains the same. *carve_object_type*$()$ returns the list of addresses (in case of online collection) or offsets (in case of offline collection) in memory containing an *Evidence_objects*. These values are then passed to the *parse_object_type*$()$ function as parameters so that the function may parse and return the list of meaningful field values (object metadata) that are of interest. Their offline counterparts *carve_object_type*$_j$*_offline* and *parse_object_type*$_j$*_offline* offer the same functionality, but their execution is deferred to after a memory dump is taken.

All these functions require a JIT-MF runtime for their implementation. Listing 4.2 presents a specification for the runtime that JIT-MF Drivers can assume and which needs to be catered for by the JIT-MF tool's main environment. *Lines 1-2* are *native*/*rt* function-hooking functions called from *init*$()$ and any other driver internal functions as needed, that enable pseudocode *lines 2,3* in Algorithm 1. *Lines 3-11* are process memory interacting functions, starting *list_memory_segments*$()$ to ensure the driver does not attempt to access un-mapped memory, or segments for which it has insufficient permissions. Memory dumping may therefore require adjusting permissions through

```
1  bool place/remove_native_hook(module,function,trigger_callback_function,[Processes]);
2  bool place/remove_rt_hook(namespace.object.method,trigger_callback_function,[Processes]);
3  [<start:address,end:address,permissions:{--|r-|rw-|rwx|...},mapped_file:string>,...]
       list_memory_segments();
4  bool set_memory_permissions(segmentbase: address, permissions : {---|r--|rw-|rwx|...});
5  [byte, ...] read_memory(at: address, length: integer);
6  bool dump_memory_segment(from: address, to: address, location: string);
7  bool dump_native_object(from: address, to: address, location: string, carve_object_type_j(),
       parse_object_type_j());
8  bool dump_rt_object(namespace.object, carve_object_type_j(), parse_object_type_j());
9  return_type  call_native_function(at: address);
10 return_type  call_rt_function(namespace.object.method,[parameters to function]);
11 bool append_log(path: string,  value: string);
```

Listing 4.2: JIT-MF Driver Runtime template.

set_memory_permissions(), as well as checking memory content through *read_memory*(). While for offline *Collection*, calling *dump_memory_segment* suffices, the driver must carve objects and parse their fields for online collection. *dump_native_object*() and *dump_rt_object* are utility functions that first locate the *Evidence_object* in memory, then execute the appropriate *carve_object_type*() and *parse_object_type*() callback functions that are passed as parameters. Separate *rt* and *native* versions are needed since the *rt* version may leverage calling runtime functions to locate the required objects. Similarly, the *native* version may leverage any memory allocators to manage native objects. *call_native_function()* and *call_rt_function*() functions allow to call existing code, at the native or runtime level respectively, that can support driver implementation, e.g., to call *mmap()* in case a scratchpad for the driver is needed. Finally, *append_log*() (*line 11*) is responsible for producing the actual *JIT-MF_Logs*, as defined in Definition 4.2.6, to the location specified by the driver's *Log_location*.

JIT-MF Driver Example. Listing 4.3 outlines a specific sample JIT-MF Driver created to collect evidence of a messaging hijack attack involving proxying of Telegram messages,[2] based on the JIT-MF Driver template in Listing 4.1, with its resulting *JIT-MF_log* shown in Listing 4.4.

```
Driver_ID: TG_CP
Scope: <telegram, crime-proxy>

Evidence_objects: {<"Telegram Message Sent","org.telegram.messenger.MessageObject",
    carve_message_object(),parse_message_object(), {"1"}>}
Collection_method: online
Parsing_method: online
Trigger_point: {<"1",<"send",native, trigger_predicate(), trigger_callback()>>}
Log_location: "/sdcard/jitmflogs"

bool init (config) {
```

[2]https://telegram.org/

```
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
}
bool trigger_predicate(params) {
  file_descriptor = params[1];
  if file_descriptor type is tcp:
    return true;
  else:
    return false;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context.args) :
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
}

[object,...] carve_message_object(from: address, to: address) {
  carve MessageObject in the given memory range using metadata provided by the
    Garbage Collector;
}

[<field,value>,...] parse_message_object(at) {
  if Parsing_method == online:
    current_time = get_time();
    MessageObject = object starting from at;

    message_content = MessageObject.messageText.value;
    message_date = MessageObject.messageOwner.date;
    message_id = MessageObject.messageOwner.id;

    recipient_id = MessageObject.recipient.id;
    recipient_name = MessageObject.recipient.name;
    recipient_phone = MessageObject.recipient.phone;

    sender_id = device_owner;
    sender_name = device_owner;
    sender_phone_number = device_owner;

    append_log(Log_location,"{'time': current_time, 'event': Evidence_objects[0][0],
     'trigger_point':Triggers[0][0], 'object':{'date':message_date, 'message_id':
    message_id, 'text':message_content,'to_id':recipient_id, 'to_name':
    recipient_name, 'to_phone':recipient_phone_number, 'from_id':sender_id, '
    from_name':sender_name, 'from_phone':sender_phone_number}}");

    return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>, <'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
}
```

Listing 4.3: JIT-MF Driver for Telegram Messaging hijack attack.

The *Evidence_object* is set (*line 4*) to Telegram's app-specific object class representing a message. The *Trigger_point* instruction is set (*line 4*) to the native system call `send`, signifying that when `send` is called to send an outgoing message to a socket (over the network or to a file). The JIT-MF Driver starts by registering the *Trigger_point* set (*line 10-14*). Once the system call registered is called to execute, the *trigger_callback()* function is called (*lines 22-30*).

The function first checks whether or not the trigger should be fired by executing the *trigger_predicate()* function. In this case, the *Trigger_point* is further filtered based on the type of file descriptor argument. Since Telegram sends messages over the network, this check is in place to ensure that the specific send function that is being intercepted comprises message-sending functionality over the network (*lines 15-21*). The *trigger_callback()* function then proceeds to dump instances of the *Evidence_object* in memory (*line 29*) using functions provided by the JIT-MF Driver Runtime. This function takes additional parameters which allow the JIT-MF Driver Runtime to carve and parse the specific object. In the example listing shown, the JIT-MF Driver is set to use *online* carving and parsing methods (*lines 5,6*) is used. As a result the *carve_message_object()* and *parse_message_object()* functions that carve and parse *Evidence_object*s execute at runtime, as soon as the *Trigger_point* returns *True* (Algorithm 1 *line 4*). The parsing of the collected *Evidence_object* requires insight into app-specific logic that defines the make-up of the *Evidence_object* (*lines37-51*). The parsed object is then appended to the *JIT-MF_log* (*line 53*) residing in `Log_location` defined in *line 8*. A sample output of the *JIT-MF_log* is shown in Listing 4.4.

4.3.3 JIT-MF installation

The JIT-MF Driver and Runtime drive and provide memory introspection capabilities through dynamic binary instrumentation, which involves loading instrumentation libraries inside the process memory that enables instrumentation. The loading of such libraries in process memory can be done dynamically or statically. A dynamic approach calls for device rooting required so that a root process can gain control over the app through *ptrace()*. Alternatively, library injection can be done statically by unpacking the app followed by minimal patching that does not affect the app's logic and placing the library inside the app's *lib/* directory. The library must then be loaded inside the process memory at runtime. While bytecode is not easily read or manipulated,

```
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614879434", "message_id": "421", "text":
    "Noise_yxUGVtS6UuShA6CuJI4lCpuP6eZ2cP5F67v", "to_id": "1168085392", "to_name": [
    contact_name], "to_name": [contact_phone], "from_id": "1679923803", "to_name": [
    owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614902759", "message_id": "723", "text":
    "Noise_hBE9b8TluuibnvHA4Fx6CDcmNdNAG5BXR52alG0Z08uyzxKME500vgyyLTrgWzYcruQCDDIYxz",
     "to_id": "961166549", "to_name": [contact_name], "to_name": [contact_phone],
    "from_id": "1679923803", "to_name": [owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614902789", "message_id": "724", "text":
    "Noise_0cHWaPUDsv7pv2PwV3QXJccLs4d4TBIx0mM6IrBnEKbqtURD1Abl0K28cDCpVyOfCN66cTBwtQ",
     "to_id": "891591776", "to_name": [contact_name], "to_name": [contact_phone],
    "from_id": "1679923803", "to_name": [owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614888830", "message_id": "552", "text":
    "Noise_lwoPcxoe1cgfyX80r2caP0LzK7G42KD9L0u3SZX1ENmExnngQ2ccFTLEMnkZrMYYfRWpn0m",
    "to_id": "891591776", "to_name": [contact_name], "to_name": [contact_phone],
    "from_id": "1679923803", "to_name": [owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614889190", "message_id": "556", "text":
    "Noise_8ytLXb6y8LXUJVnAbVhQvjGE0TFHQAFD2Gj4ji4PJN", "to_id": "891591776", "to_name"
    : [contact_name], "to_name": [contact_phone], "from_id": "1679923803", "to_name": [
    owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614889324", "message_id": "558", "text":
    "Noise_KELADQloIFUVGHqM4JNtd42DH", "to_id": "891591776", "to_name": [contact_name],
     "to_name": [contact_phone], "from_id": "1679923803", "to_name": [owner_name],
    "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614890015", "message_id": "561", "text":
    "Noise_jaWWAqo3e83MsIZ8bCZop7bBviqEepR5J3P7ecUllD5frTsiv7OPdiVIhxwAK0PE6xjEDTRlxu",
     "to_id": "891591776", "to_name": [contact_name], "to_name": [contact_phone],
    "from_id": "1679923803", "to_name": [owner_name], "from_phone": "[owner_phone]"}}
{"time": "1614902986","event": "Telegram Message Sent","trigger_point": "native",
    "object": {"date": "1614890459", "message_id": "565", "text":
    "Noise_KIBWB2GNH9knGog", "to_id": "891591776", "to_name": [contact_name], "to_name"
    : [contact_phone], "from_id": "1679923803", "to_name": [owner_name], "from_phone":
    "[owner_phone]"}}
```

Listing 4.4: Sample of *JIT-MF_log* containing *Evidence_object* from a Telegram messaging hijack scenario.

an intermediate language Smali, can be used to patch the code. The library must be loaded early in the app's lifecycle to ensure that all functions executed during the app's runtime can be instrumented. Therefore, the class within the app implementing the `MainActivity`, which executes when the app is launched and is listed in the app's Manifest file, is patched. Its Smali code, found in the decompiled files of the app, contains a static constructor, which is modified to include the Smali equivalent of the function call: *System.loadLibrary("instrumentation-library")*.

A static approach is preferred to abide by the requirements set out for this solution and avoid device rooting. Yet, the static injection of an instrumentation library inside an app, as required by DBI frameworks, still calls for an app-invasive solution that involves app repackaging. Alternatively, app-level virtualisation can emulate a runtime for the targeted app, which loads the instrumentation library before loading the app through a customised APK loader. Therefore, app-level virtualisation allows JIT-MF to be installed and operate in a minimal app and device-invasive approach that does not involve app-repackaging or device rooting. Yet, given that app-level virtualisation is still an emerging technology, its use exposes concerns regarding the effects on performance and app stability. Experimentation regarding the further exploration of JIT-MF Driver properties and the value of JIT-MF in forensic investigations used both approaches.

4.3.4 JIT-MF Driver Runtime technology enabler

The JIT-MF Driver Runtime exposes the required services for the JIT-MF Driver to function and produce *JIT-MF_Logs*. The critical service that the Driver Runtime exposes is registering *Trigger_point*, enabling timely dumping of *Evidence_object*s. A primary enabler for this is dynamic binary instrumentation.

Frida [50] is a dynamic binary instrumentation toolkit that allows code instrumentation on Android through inline hooking. It implements Xposed-style method hooking and native inline hooking, enabling functions invoked from native or bytecode (by the app at runtime) to be instrumented. Thus enabling function calls from programs at any layer of the Android technology stack to be registered as *Trigger_point*s. Frida, operates in multiple modes: injected, preloaded and embedded. The embedded mode offers an option that aligns more with the requirements set out for JIT-MF. Frida's embedded mode, known as Frida Gadget,[3] allows for hooking through a shared library (*frida-gadget.so*) that can be loaded inside apps, either through app repackaging or a virtualised environment using a customised APK loader.

The memory introspection capability required by the JIT-MF Driver Runtime (Listing 4.2 *lines 1,2*) can be satisfied by Frida's Java API built-in module.[4] This module allows interoperability with the Java runtime, and exposes functionality which enumerates live instances of the Java classes in memory through the exported *Heap::GetInstances* function in *libart.so*. This suffices to enable the online collection and carving of evidence objects from memory required by JIT-MF Drivers (listed as separate functions in Listing 4.2 *lines 3-8*). The same interoperability with the Java runtime allows for the persistence of evidence to storage through the File module[5] that writes to file streams. Dumping the entire process memory space (for offline collection) is also possible by calling the Java runtime function *android.os.Debug.dumpHprofData()* (as shown in Listing 4.2 *line 10*). This

[3]`https://frida.re/docs/gadget/`
[4]`https://frida.re/docs/javascript-api/#java`
[5]`https://frida.re/docs/javascript-api/#file`

functionality is sufficient to provide the services the JIT-MF Driver Runtime requires. Furthermore, Frida allows custom scripts (written in Javascript) that drive the instrumentation process to be loaded at runtime. Thus, JIT-MF Drivers can be implemented as supplementary scripts, defining which specific definitions for *Evidence_object*s and *Trigger_point*s that are custom to specific scenarios and are loaded at runtime. Therefore, this toolkit is used as a starting point for implementing the JIT-MF Driver Runtime and, consequently, the JIT-MF Drivers that use it.

4.4 Operational context

JIT-MF Driver developers develop JIT-MF Drivers to be used within JIT-MF tools to produce JIT-MF_Logs for a specific app and attack scenario. The process by which a JIT-MF Driver developer develops a JIT-MF Driver is shown in Figure 4.2. The *Evidence_object* is set as an initial step through knowledge of the app and the hijack scenario. In the case of messaging hijack attacks, *Evidence_object* constitute message objects which may have been generated by the app while under the influence (hijacked) by malware.

The *Trigger_point* of a JIT-MF Driver can be set based on the selected *Evidence_object* and knowledge of the context (operating system and investigation scenario) within which the driver will operate. The JIT-MF Driver developer then must decide whether evidence carving will occur at runtime (online) or at a later stage (offline). Online evidence carving would require the JIT-MF Driver runtime to provide an enabler by which app-specific *Evidence_object*s can be carved out from memory in real-time through mechanisms like garbage collectors. Offline evidence carving means that any parsing of the object metadata into meaningful information that can later be correlated and used for analysis must be carried out offline. When object carving is set to be online, the JIT-MF Driver developer must decide whether evidence parsing will occur at runtime (online) or after the evidence has been dumped (offline). This decision is based on the amount of processing required for parsing the evidence object and the developer's insight into the impact on the stability of the setup that this additional parsing will have.

Chain of custody. The JIT-MF Driver developer needs to know the context within which the evidence generated will be used. In investigations involving attack scenarios requiring the involvement of legal entities and law courts, the integrity of the evidence generated plays a crucial role. A proper chain of custody ensures that regardless of the process involved in collecting, transporting and handling the evidence, the integrity of the evidence can be verified. This is typically done by using hashing functions that

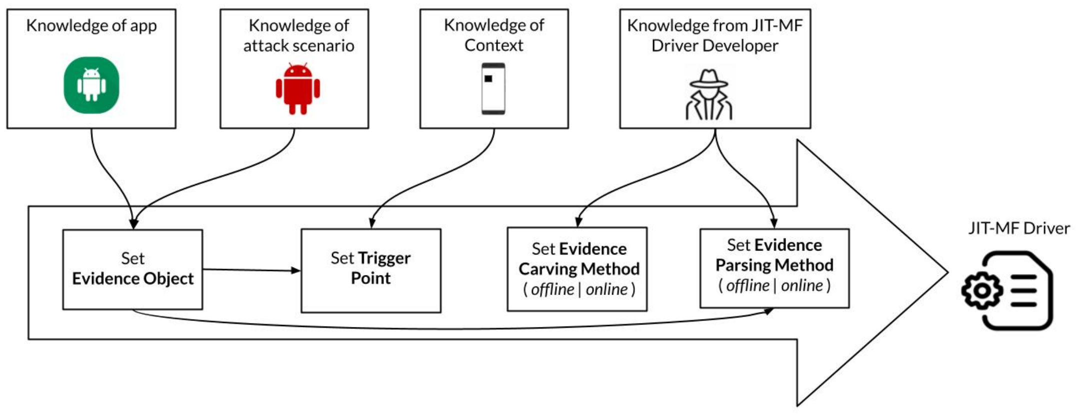

Knowledge of app
Knowledge of attack scenario
Knowledge of Context
Knowledge from JIT-MF Driver Developer
Set Evidence Object
Set Trigger Point
Set Evidence Carving Method (offline | online)
Set Evidence Parsing Method (offline | online)
JIT-MF Driver

Figure 4.2: Process for developing a JIT-MF Driver.

allow investigators to verify that the hash value of the evidence remains the same. For evidence collected through a JIT-MF Driver, the JIT-MF Driver developer must implement the functionality to keep a digital chain of custody that can be used to verify the integrity of the *JIT-MF_Logs* produced. This chain of custody implementation would need to record the steps carried out throughout each stage of the JIT-MF tool. At each step, the hash values for any outputs generated (such as enhanced apps, *JIT-MF_Logs*) are stored alongside the hash of the JIT-MF Driver itself. The same applies to any parsing functionality or tool that processes the JIT-MF_Logs once they are produced and collected [36].

***JIT-MF_Logs* collection.** The resulting *JIT-MF_Logs* produced by JIT-MF, per instructions inside JIT-MF Drivers, are stored on the device by default. Therefore another concern for the JIT-MF Driver developer is the accumulation of these logs on the device, which may exhaust the storage resources of the device. To avoid such a scenario, the JIT-MF Driver developer must be aware of the owner usage patterns of the targeted app usage and resulting *JIT-MF_Logs* size. Following this analysis, the JIT-MF Driver developer can devise a plan for scheduled extraction of *JIT-MF_Logs* to be stored on separate storage. This ensures that the device resources are not exhausted while the *JIT-MF_Logs* are retained in case an investigation is needed.

JIT-MF Drivers are used by JIT-MF tools built on the JIT-MF framework. Any JIT-MF tool comes into play at the forensic readiness stage [75] to forensically enhance Android devices before an incident is flagged. Figure 4.3 shows how a JIT-MF tool fits within the incident response cycle stages. At the forensic readiness stage, targeted users and their devices and apps are identified during an asset management exercise (step 1). These

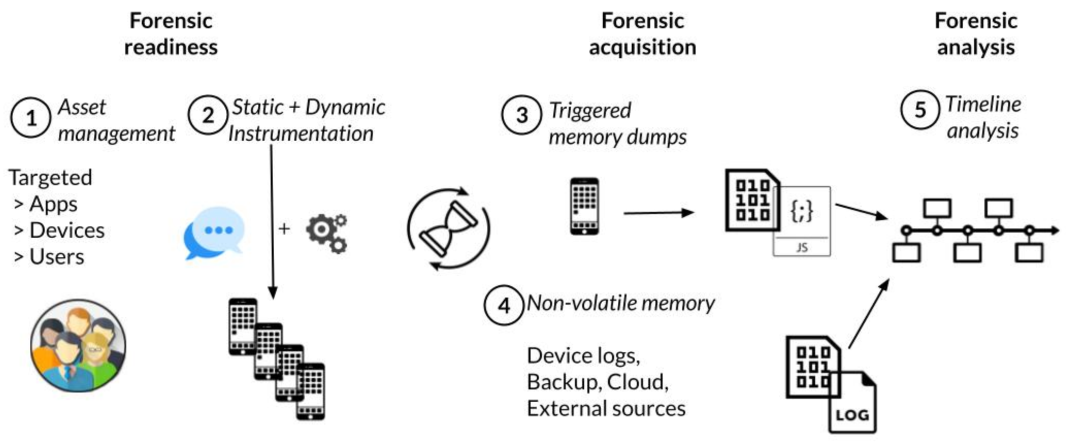


Figure 4.3: JIT-MF within the Incident Response cycle.

users can be high-profile employees of government agencies or even private citizens whose devices may be the target of resourceful attackers for various reasons. After this stage, those apps that pose a particular risk, say messaging apps, are instrumented post-deployment with JIT-MF Drivers and Driver Runtime (step 2). While the app executes, forensic artefacts are collected from memory inside *JIT-MF_Logs*, based on the properties defined in the JIT-MF Drivers (step 3). Once suspicious activity is noticed, with alerts possibly raised by the device owners or by incident responders during routine checks, *JIT-MF_Logs* can be merged with other forensic sources to produce a more comprehensive forensic timeline (steps 4 and 5). A JIT-MF tool automates this workflow.

4.5 Security analysis of JIT-MF

JIT-MF_Logs are by default stored on the device and can potentially contain sensitive information. Therefore these logs expose an attack surface that malware could exploit. Should these logs fall prey to any malware on the device, these logs may be subject to anti-forensics techniques that tamper or delete *JIT-MF_Logs*. Crucially, if these logs are accessible by the malware, sensitive data may be exfiltrated by the malware. Similarly, but from an operational point-of-view, JIT-MF Drivers are also stored on the device to instrument a targeted app. Therefore, these are also susceptible to tampering by malware with malicious intent, with the result being total control over the targeted app. While these issues are of concern in a real-world setting, mitigations for these attacks surfaces are not an open research problem. Existing measures and techniques can be used to protect JIT-MF usage in practice.

Privacy-aware forensics solutions exist [45] through which the proposed JIT-MF Drivers can collect the necessary evidence to reconstruct stealthy attack steps while protecting sensitive information to protect users' privacy. The use of work profiles is recommended for enterprise settings,[6] which respect employee privacy through separate, dedicated work and personal profiles that give investigators more flexibility concerning privacy when retrieving data related to work profiles. Finally, scoped storage can appropriately store *JIT-MF_Logs* and Drivers, thus ensuring secure access to these critical contents.

Other adversarial tactics can include the exploitation of JIT-MF for resource exhaustion. JIT-MF-aware malware may purposefully invoke *Trigger_point*s to produce large enough *JIT-MF_Logs* files that exceed the resource capacities of the device while rendering the app unusable due to constant execution of *Trigger_point*s. On the other hand, JIT-MF-aware malware may use adversarial tactics to avoid leaving a trail of evidence. Similarly to API unhooking techniques used to evade EDR detection [13, 79], the malware developer may attempt to avoid invoking specific instructions that constitute trigger points during malware execution. This is difficult for app hijack attacks to do since this functionality is specific to the app being hijacked and, therefore, out of the malware's reach. The alternative for the malware developer is hijacking another app on the device that provides the malware with similar functionality but has not yet been enhanced with a JIT-MF Driver.

JIT-MF Driver developers can mitigate JIT-MF-aware malware resource exhaustion attempts by introducing frequency checks in *Trigger_point* predicates within the JIT-MF Driver. These frequency checks can rate-limit the dumping of *Evidence_object*s, thus avoiding exceeding the device's resource capacity. Furthermore, at a forensic readiness stage, all apps on the device owner's phone that have the potential to be hijacked due to their functionality should be enhanced with a JIT-MF Driver to ensure evidence is still collected regardless of the hijacked app.

4.6 Summary

This chapter presented Just-in-Time Memory Forensics (JIT-MF), a framework aiming to provide a blueprint for implementing tools that timely collect app-specific objects from volatile memory which can be linked to app hijack attack steps. The framework's main elements comprise: i) *Evidence_object*s, the app-specific objects whose presence in memory implies the execution of some specific app functionality, possibly a hijacked attack step; ii)

[6]`https://www.android.com/enterprise/work-profile/`

Trigger_point, the instructions associated with the presence of *Evidence_objects* in memory which invoke a memory dump; and iii) *JIT-MF_Logs* which is the output comprising *Evidence_objects* produced from the triggered memory dump (via *Trigger_point*).

Dynamic binary instrumentation is identified as the primary enabler for timely evidence collection from memory. *Trigger_point*s and *Evidence_object*s are defined within JIT-MF Drivers that drive the evidence collection process from memory for a specific app hijack scenario. While JIT-MF Driver Runtime provides the necessary services required by the Driver, namely process memory introspection and carving objects from memory.

The requirements set out in the beginning of this chapter aim to address the invasiveness challenges highlighted in this thesis' research question. This chapter described how the installation of JIT-MF can be minimally invasive thus compyling with the requirements set out at the beginning of this chapter, by using app-level virtualisation to instrument apps while avoiding app repackaging and device rooting. However, the research question is also concerned with the level of invasiveness requried for code comprehension effort, to identify app-specific artefacts related of app hijack attack steps. Thus, an exploration of *Evidence_object* and *Trigger_point* selection within the Android technology stack is required to determine the level of app invasiveness necessary for JIT-MF tools to collect accurate evidence. This can be explored through the application JIT-MF in realistic case studies involving app hijack attacks. The next chapters explore the minimum level of app-level invasion possible without compromising forensic timeline accuracy and comparisons to the state-of-the-art.

5 Exploration of JIT-MF positioning within the Android Technology stack

The previous chapter proposed Just-in-Time Memory Forensics (JIT-MF) as a means to explore collecting app-specific evidence from memory, uncovering the execution of hijacked app functionality in a manner that is timely (that is before evidence from memory is lost) and not prone to forensic trace deletion. This is achievable through the definition and selection of JIT-MF *Evidence_object*s and *Trigger_point*s as set in JIT-MF Drivers, provided the functionality of the underlying JIT-MF Driver Runtime. The result is timely collected app-specific artefacts from memory stored in *JIT-MF_Logs*. Yet further exploration is still required to determine how the tools implementing this framework can meet the minimal invasiveness requirements of this thesis' research question; that is, ensuring compatibility with stock devices while also being minimally invasive at the app and device levels to produce accurate forensic timelines. Specifically, this chapter aims to test the hypothesis by exploring which layer within the Android technology stack is the optimal layer in terms of accuracy and minimal app invasiveness through the selection of JIT-MF *Trigger_point*s and *Evidence_object*s, thus addressing the second objective ***O2*** of this thesis.

The experimentation in this thesis focuses on messaging apps as the targeted benign apps of app hijack attacks. Messaging is a current concern regarding the sensitivity of its usage (Section 5.1). An initial experiment (Section 5.2) is carried out to establish the difference in the accuracy of evidence produced in *JIT-MF_Logs* when selecting *Trigger_point*s from different layers in the technology stack contributing to different levels of app invasiveness. Once the optimal *Trigger_point* selection layer is established concerning both invasiveness and accuracy, a follow-up experiment (Section 5.3) aims to establish whether *Trigger_point*s at this layer can help produce more accurate forensic timeline sequences with sufficient detail, when used in conjunction with commonly-used open source forensic sources. An exploration of *Evidence_object* selection, concerning the varying layers of the stack, is also carried out similarly to *Trigger_point* selection exploration, aiming for an even more minimally invasive approach while retaining accuracy (Section 5.4). Finally, the effects on app stability are considered (Section 5.5) and possible solutions are evaluated regarding app stability and accuracy.

5.1 Methodology

The experiments carried out to explore JIT-MF positioning across the Android technology stack require a shared methodology regarding experimentation setup. Mainly, this concerns the attack scenarios used in the experiments and the JIT-MF Driver and Runtime prototype and installation used for experimentation.

5.1.1 Attack scenarios

The experiments comprise a suite of messaging hijack case studies inspired by real-life app hijack attacks that target messaging apps [7]. The messaging hijack scenarios that are considered involve: i) crime-proxying; and ii) unlawful interception. Crime-proxying refers to app hijack attacks that hijack the messaging functionality of Android devices to hide compromising communication of a criminal nature behind victim devices. This can be done by hijacking a victim's messaging app functionality to proxy messages with incriminating content and deleting them immediately after to avoid suspicion from the victim. Unlawful interception refers to spying on a victim's incoming and outgoing messages. In these scenarios, app functionality of interest comprises message-sending and reading functionality, respectively. Benign apps were selected during experiments to simulate the targeted apps to be hijacked. These apps were chosen based on their functionality as SMS or Instant Messaging (IM) apps.

Simulations of these attacks were implemented following the app hijack threat model, using the accessibility attack vector. A separate attack was created per targeted app and attack scenario (crime-proxy or spying). Various implementation methods were used to implement these attacks, according to the aims and automation requirements of the experiment. Some attacks were implemented in the Metasploit pentest framework, as part of the Android Meterpreter payload. Once the idea of stealthy attack capabilities was rendered, simulations of the same attack were carried out using `adb` via the `AndroidViewClient`[1] (for ease of automation), emulating any existing or future attack vector through which app functionality can be hijacked.

[1]`https://github.com/dtmilano/AndroidViewClient`

5.1.2 JIT-MF prototype and installation

JIT-MF was installed at the app level to ensure the use of stock devices (as per the first requirement of JIT-MF). All case studies for the experiments described in this chapter involved installing JIT-MF Driver and Driver Runtime via app repackaging as the exploration described in this chapter focuses on the level of app-invasiveness required for selecting *Trigger_point*s and *Evidence_object*s. For the case studies used during this exploration, application functionality of interest comprised message-sending and reading functionality that app hijack malware can hijack for crime-proxying and spying purposes, respectively.

The JIT-MF Driver runtime was provided by a subset of the Frida[2] runtime, and JIT-MF Drivers were implemented as Javascript code for Frida's Gadget shared library. The JIT-MF Drivers created for each case study, based on the JIT-MF Driver template, and resources for experiment setup can be found in the associated open source repository[3].

5.2 Trigger point accuracy across Android stack layers

Selecting *Trigger_point*s and *Evidence_object*s from higher levels in the Android technology stack requires increasingly app-invasive approaches, that impose infeasible development effort unique to each app. Specifically, this requires increased and individual effort per app in comprehending its codebase, which is likely obfuscated and close-source. Regarding the layer for *Evidence_object*s selection, the app layer presents the most obvious choice in the Android technology stack. This layer comprises app-specific artefacts that can be directly linked to attack steps of attacks following the app hijack threat model, thus having the potential to generate the most accurate forensic timelines. However, it is possible that the optimal layer for *Trigger_point* position can be found in lower layers of the stack, resulting in a less app-invasive approach.

The experiment described in this section compares the accuracy of the attack steps recorded and the associated storage overhead costs when selecting *Trigger_point*s from different layers in the Android technology stack. The same *Evidence_object*s per app were selected from the app layer since this layer presents the most obvious choice for producing app-specific evidence objects that can be linked to hijacked app functionality and therefore used to reconstruct attack steps on a forensic timeline accurately.

[2]`https://frida.re/docs/android/`
[3]`https://gitlab.com/bellj/dissertation_resources`

5.2.1 Experiment setup

Pushbullet[4] and Telegram[5] are popular SMSonPC and IM apps, respectively, used as targeted apps in case studies for this experiment. The experiment involves four case studies comprising messaging hijack scenarios involving spying and crime-proxying attack scenarios hijacking the two aforementioned targeted benign apps.

All four attack scenarios were implemented as extensions to Metasploit's Meterpreter for Android[6]. For the Telegram IM case study, the attack was carried out using the Android Metasploit attack suite. App hijack attack on Pushbullet, being an SMSonPC app allowing for remote messaging, comprised using phished credentials. The remaining attack steps to send messages made direct use of Pushbullet's remote web portal and were automated using Selenium[7] whereas any incoming messages could be obtained from browser logs. The full setup comprises: Pushbullet (v17.7.19) and Telegram (v6.1.1) Android apps enhanced with JIT-MF Drivers, both installed on an Android 10 emulator. The JIT-MF Drivers used implemented both online and offline evidence collection methods (as described in Section 4.3.1), leveraging Frida's *Java.choose()* and Android's API *Debug.dumpHprofData()* respectively. The attack scenarios were repeated ten times since it sufficed to reach convergence for all measurements taken.

Eight JIT-MF Drivers were created per case study. The chosen *Evidence_object*s listed in Table 5.2 were selected from the app stack layer (for each app), yet in both cases, the object was easily discernible from the documentation. In Telegram's case, the app is open source and available on Github[8]. Therefore identifying the *Evidence_object* required inspecting the code and using search prompts that could identify the object holding a Telegram message along with available documentation. Pushbullet is closed-source. However, it exposes an interface that enables users to send messages over HTTPS. The related documentation and network traffic analysis during app usage enabled the *Evidence_object* selection. From preliminary analysis, three stack layers were identified as promising candidates for *Trigger_point* positioning to start experimentation: the App, API layer and Native layers as shown in Table 5.1. Two *Trigger_point*s were selected from each layer, attempting to leverage all available candidate *Trigger_point*s in terms of disk input/output, network send/receive, and miscellaneous object transformations. The native layer was further leveraged for experimentation using *Trigger_predicate*(). For this

[4]https://www.pushbullet.com/

[5]https://telegram.org/

[6]https://github.com/rapid7/metasploit-framework/tree/master/documentation/modules/payload/android

[7]https://selenium-python.readthedocs.io/

[8]https://github.com/DrKLO/Telegram

layer, four *Trigger_point*s were selected comprising functions from the native *libc* library. Two were combined with native-level predicates, while the others had device-level predicates triggered by generic device events, such as increased network traffic. While *Trigger_point*s at the API level could have sufficed for creating *Trigger_point*s that leverage device events, for example, *HttpURLConnection.connect()*, native layer function calls *recv* and *send* represent a chokepoint for multiple functions at the API level, and therefore is simpler to work with. The chosen list of *Trigger_point*s is presented in Table 5.3, where the first *Trigger_point* (TP1) is either file/disk or object transformation-related, and the second (TP2) is network-related.

Table 5.1: Android stack layers associated trigger predicates identified for exploration of *Trigger_point* positioning and classification in terms of app internals comprehension required.

Stack layer	*Trigger_predicate*()	**Classification**	**Description**
App	-	Most app-invasive	Function calls specific to the app
APIs (Android & third-party)	-	Least app-invasive	Android API calls
Native	Native-level	Least app-invasive	Generic native system calls
Native	Device-level	Least app-invasive	Generic events related to the device state

Table 5.2: *Evidence_object*s selected per app.

App	*Evidence_object*
Pushbullet	`org.json.JSONObject`
Telegram	`org.telegram.messenger.MessageObject`

Trigger point accuracy comparison. Accuracy of the *Trigger_point* selected is measured by identifying whether or not the proxied or stolen messages (depending on the hijack scenario) can be found in the *JIT-MF_Logs* produced when the respective *Trigger_point*, selected from a specific stack layer.

Performance overheads. An initial analysis is carried out to measure the potential runtime performance overheads in storage and execution time. These performance indicators were measured for both apps while application functionality of interest was executing: message sending and reading/retrieving during which JIT-MF *Trigger_point*s are invoked.

Table 5.3: Trigger points selected.

Case Study	Stack layer	TP #	Instruction	Trigger Predicate
(SMS) Pushbullet - Crime-proxy	App	TP 1	`com.pushbullet.android.sms.SmsSyncService.a`	-
		TP 2	`com.pushbullet.android.providers.syncables.SyncablesProvider.insert`	-
	API	TP 1	`android.content.ContentResolver.insert`	-
		TP 2	`android.telephony.SmsManager.sendTextMessage`	-
	Native (combined with native-level *Trigger_predicate*())	TP 1	`write()`	To file descriptor
		TP 2	`read()`	From TCP Socket as file descriptor
	Native (combined with device-level *Trigger_predicate*())	TP 1	Increase in app directory size	`app_directory_path=Context().getFilesDir().getParent()`
		TP 2	Increase in network traffic	Incoming
(SMS) Pushbullet - Spying	App	TP 1	`com.pushbullet.android.sms.SmsSyncService.a`	-
		TP 2	`com.pushbullet.android.gcm.GcmService.a`	-
	API	TP 1	`android.content.ContentResolver.registerContentObserver`	-
		TP 2	`com.google.android.gms.gcm.GcmReceiver.onReceive`	-
	Native (combined with native-level *Trigger_predicate*())	TP 1	`read()`	From file descriptor
		TP 2	`write()`	To TCP Socket as file descriptor
	Native (combined with device-level *Trigger_predicate*())	TP 1	Increase in app directory size	`app_directory_path=Context().getFilesDir().getParent()`
		TP 2	Increase in network traffic	Outgoing
(IM) Telegram - Crime-proxy	App	TP 1	`org.telegram.tgnet.ConnectionsManager.native_sendRequest`	-
		TP 2	`org.telegram.messenger.SendMessagesHelper.performSendMessageRequest`	-
	API	TP 1	`android.widget.EditText.setText`	-
		TP 2	`android.app.SharedPreferencesImpl\$EditorImpl.commitToMemory`	-
	Native (combined with native-level *Trigger_predicate*())	TP 1	`open()`	File descriptor
		TP 2	`send()`	To TCP Socket as file descriptor
	Native (combined with device-level *Trigger_predicate*())	TP 1	Increase in app directory size	`app_directory_path=Context().getFilesDir().getParent()`
		TP 2	Increase in network traffic	Outgoing
(IM) Telegram - Spying	App	TP 1	`org.telegram.ui.Cells.DialogCell.update`	-
		TP 2	`org.telegram.messenger.MessagesStorage.putMessages`	-
	API	TP 1	`android.view.ViewGroup.dispatchGetDisplayList`	-
		TP 2	`android.app.ContextImpl.sendBroadcast`	-
	Native (combined with native-level *Trigger_predicate*())	TP 1	`open()`	File descriptor
		TP 2	`recv()`	From TCP Socket as file descriptor
	Native (combined with device-level *Trigger_predicate*())	TP 1	Increase in app directory size	`app_directory_path=Context().getFilesDir().getParent()`
		TP 2	Increase in network traffic	Incoming

Table 5.4: Trigger point effectiveness: % accuracy over ten runs. Values highlighted in grey represent the *Trigger_point*s that performed best in layers (with associated trigger predicates) that fall under the least app-invasive classification, per investigation scenario.

Trigger point classification & Stack layer (with associated trigger predicates / Scenario			**Crime-proxy - IM**		**Spying - IM**		**Crime-proxy - SMS**		**Spying - SMS**	
			Online	Offline	Online	Offline	Online	Offline	Online	Offline
Most app-invasive	App	TP 1	100	100	100	100	0	0	0	0
		TP 2	0	0	60	60	100	100	80	80
Least app-invasive	API	TP 1	90	90	100	100	100	100	0	0
		TP 2	80	60	100	100	100	100	80	80
	Native (combined with native-level *Trigger_predicate*())	TP 1	100	80	100	100	30	80	80	80
		TP 2	50	50	100	100	100	100	0	30
	Native (combined with device-level *Trigger_predicate*())	TP 1	40	40	100	90	50	80	20	30
		TP 2	50	50	100	100	100	100	0	30

5.2.2 Results

Table 5.4 compares the *Trigger_point*s based on accurately dumping *Evidence_object*s related to the proxied or intercepted SMS/IM messages over ten runs per attack. The first two rows are the results obtained for the most app-invasive *Trigger_point*s (selected from the app layer), while the next six rows are for the least app-invasive *Trigger_point*s. The results column-wise (across *Trigger_point*s) from different stack layers, show that the assumption that app-specific and app-invasive *Trigger_point*s positioned at the app layer are more accurate than less app-invasive counterparts does not hold. For each of the case studies, the collected evidence contains the following metadata: i) the contents of the message sent/read; ii) the sender/recipient (for crime proxy and spying, respectively); and iii) the time at which the message was received/intercepted.

At first glance, it seems that selecting accurate *Trigger_point*s could be possible solely within the layers requiring the least app-invasive approaches and minimal app-specific knowledge. Furthermore, the results presented in Table 5.4 also show that the effectiveness obtained using offline and online collection methods has very similar results.

Least app-invasive *Trigger_point*s show promise. The fact that *Trigger_point*s selected from lower levels in the technology stack can be as effective and efficient as those selected from the app layer bodes well for the second requirement set out for JIT-MF, i.e. ensuring a minimal app invasive approach, since lower layers of the stack require less code comprehension effort than app-specific compiled code analysis. Yet comprehension of app functionality remains a factor for selecting *Evidence_object*s since these have so far been selected from the app layer.

Considering only the best-performing *Trigger_point*s within each of the least app-invasive layers and associated trigger predicates, per investigation scenario (highlighted in grey), it is noticeable that across the four scenarios, *Native (combined with device-level Trigger_predicate())* *Trigger_point*s were the least effective, with the rest at a tie. Furthermore, in the cases where *Trigger_point*s positioned at the *Native layer (combined with device-level Trigger_predicate())* underperformed, the discrepancy was substantial, e.g. in the *Spying - SMS* case, 60% fewer events were caught when compared to the results obtained by the best-performing *Trigger_point*s in the other two least app-invasive layers. These results, therefore, suggest that in the case of *Native (combined with device-level Trigger_predicate()) Trigger_point*s, several *Trigger_point*s may have to be evaluated to determine their effectiveness before deployment. While for certain app functionality, one can assume underlying app events (e.g. `send` syscall in case of an outgoing message over the network), this functionality's effects on the devices may differ depending on app usage and the device itself. For instance, if an app stores data in a cache store until a limit is reached and then empties the cache, selecting a *Native (combined with device-level Trigger_predicate())* *Trigger_point* that monitors an increase in app directory size may not be sufficient since the frequency with which the device owner is sending messages hinders the effectiveness of said *Trigger_point*. Therefore gaining access to a typical app usage profile becomes valuable in assisting the driver developer in generating better *Native (combined with device-level Trigger_predicate())* *Trigger_point*s targeted to a scope.

Furthermore, considering the *Trigger_point*s positioned in the two most successful stack layers and associated trigger predicate under the least app-invasive classification (*Native (combined with native-level Trigger_predicate())* and *APIs*), results show that the distinction in effectiveness between file/disk-related *Trigger_point*s (TP1) and network-related *Trigger_point*s (TP2), remains unclear. This is evident as results vary substantially between the two types of *Trigger_point*s even in the same stack layer and scenario. That said, network-related fared consistently in the scenarios involving the Telegram IM app. The scenarios involving SMS show mixed results, demonstrating that even within the context of message hijack attacks, different types of *Trigger_point*s leveraging both file/disk and network might be required depending on the type of messaging app involved in the scenario.

Overhead performance costs. Given that *Trigger_point*s positioned at the least app-invasive layers can be as effective as those in the app layer, the focus shifts onto the performance overheads these *Trigger_point*s incur since these *Trigger_point*s are more aligned with the requirements set out for JIT-MF. Considering only the best-performing *Trigger_point*s within each of the least app-invasive layers and associated trigger predicates, per investigation scenario (highlighted in grey), the results show that these

Table 5.5: Trigger point performance overheads: Additional kB required in storage and execution time (in seconds). The results in this table reflect overheads for the most effective *Trigger_point*s during online collection.

Trigger point classification & Stack layer (with associated trigger predicates / Scenario	**Crime-proxy - IM**		**Spying - IM**		**Crime-proxy - SMS**		**Spying - SMS**	
	Average storage (kB)	Average time increase (s)	Average storage (kB)	Average time increase (s)	Average storage (kB)	Average time increase (s)	Average storage (kB)	Average time increase (s)
App	45.704	-	10.234	-	1.857	0.1068	0.611	0.1068
API	43.383	-	10.363	-	0.1384	0.0669	2.762	0.0669
Native (combined with native-level *Trigger_predicate*())	61.166	-	11.133	-	20.054	0.8265	90.228	2.4885
Native (combined with device-level *Trigger_predicate*())	126.06	-	40.657	-	30.381	0.3026	46.95	2.9532

- Average increase in execution time of app functionality of interest is negligible.

*Trigger_point*s do not necessarily incur higher storage costs, with online collected dumps requiring as little as 0.1kB to be effective for one case study. However, these results must also be analysed in the context of practical JIT-MF tool deployment in a realistic setting. When one considers that dumps are triggered per critical app functionality, corresponding to SMS/IM sending/viewing in the experiment case studies, dumps are expected to be very frequent. While perhaps SMS is less concerned nowadays, IM is an entirely different story that could result in daily triggers on the order of hundreds to thousands, even in the case of online collected dumps.

The effectiveness of offline and online collection methods have very similar results. Yet, it is of note that the average dump size required by online collection is around 143kB. In contrast, that required by the offline method is 203MB (three orders of magnitude *more* on average), per attack scenario and *Trigger_point* chosen. Therefore, the offline collection method becomes less preferred due to the size of the resulting *JIT-MF_Logs* on the device. As a result, the following analysis and experiments consider online collection.

Table 5.5 shows the average storage cost and additional execution time incurred (optimal in case TP1 and TP2 are equally effective), for the best-performing *Trigger_point*s considering only online collection. This table shows that execution overheads incurred while app functionality of interest executed were negligible in Telegram's case. For Pushbullet, this value increases to 2.9s at worst. However, given that Pushbullet operates from a browser setting, this execution overhead does not incur any lag on the phone's main UI thread, enabling the user to continue using the phone normally. Moreover, while both effectiveness and runtime overheads so far do not overwhelmingly favour any of the layers which are least app-invasive across the board, it seems that optimal performing least app-invasive *Trigger_point*s selected from certain layers of the stack (specifically those at the API layer) might be less resource-intensive than others, *Trigger_point*-wise. This merits further field analysis and possible mitigation, which is explored in Section 5.5.

5.3 Accuracy of forensic timelines using *JIT-MF_Logs*

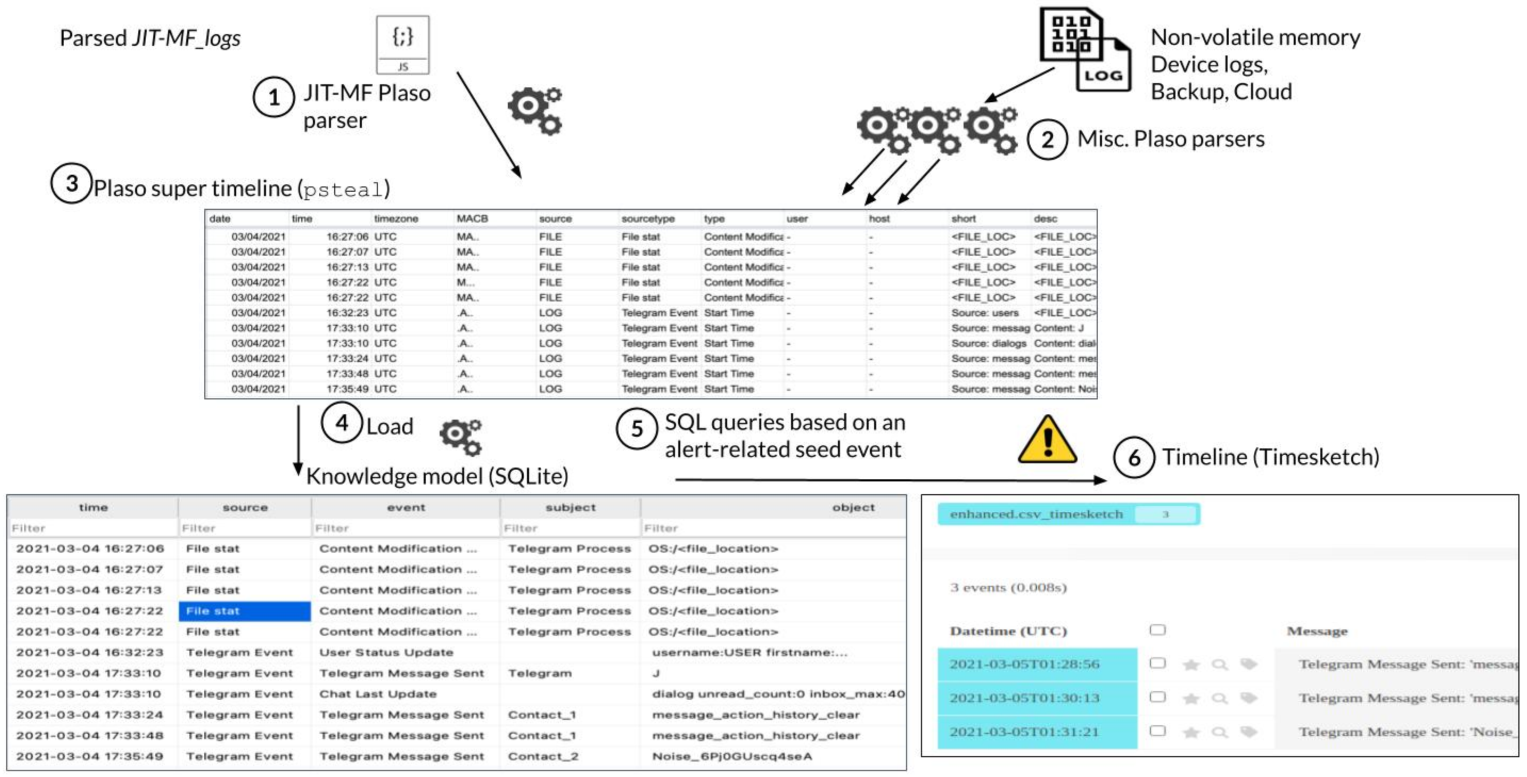
The previous experiment demonstrated that the accuracy of app artefacts found in *JIT-MF_Logs* is not impacted when selecting *Trigger_point* from lower layers of the Android technology stack, which require a less app-invasive approach in terms of layer internals comprehension. This offers a basis upon which *Trigger_point*s are selected from this point onwards. While the accuracy of *JIT-MF_Logs*, based on *Trigger_points* selection from different stack layers, has been observed from experimentation results, the accuracy that these *Trigger_point*s provide in terms of forensic timeline generation for investigations of app hijack attacks, has yet to be determined. This experiment aims to determine the accuracy of the forensic timelines generated in the case of messaging hijack attacks when using *JIT-MF_Logs* produced by JIT-MF Drivers with minimally invasive *Trigger_point*s as an additional forensic source of evidence to the forensic sources typically used by forensic tools.

A series of case studies involving messaging hijack attacks are simulated to assess the added accuracy of the newly-created forensic timelines. For each case study, a comparison is made between the recorded app-specific artefacts found in *JIT-MF_Logs* (in the form of *Evidence_object*s) and the ground truth attack steps executed as part of the simulated attacks. Further comparison is made to assess further how the timeline generated using *JIT-MF_Logs* as an additional forensic source compares to typically generated forensic timelines that do not include *JIT-MF_Logs*, which are referred to as *baseline sources*.

5.3.1 Forensic timeline generation

Forensic timeline generation considers all forensic sources that can shed light on app usage. These range from the device-wide `logcat` to app-specific sources inside `/data/data`, as well as inside removable storage which can be found in the `sdcard` partition and whose mount point is device-specific. Evidence collection from the device is opted for, rather than cloud or backups, to facilitate experimentation whenever the same data could be obtained from multiple sources (see Section 2.2). These forensic sources represent those collected by state-of-the-art mobile forensics tools, typically requiring device rooting or a combination of hardware-based physical collection and content decryption. These baseline sources can be complemented by *JIT-MF_Logs* if they do not produce the necessary evidence for hijacked attack steps.

Figure 5.1 shows the process that transforms the evidence obtained from the aforementioned forensic sources into finished forensic timelines. This pipeline is based on



date	time	timezone	MACB	source	sourcetype	type	user	host	short	desc
03/04/2021	16:27:06	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC>
03/04/2021	16:27:07	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC>
03/04/2021	16:27:13	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC>
03/04/2021	16:27:22	UTC	M...	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC>
03/04/2021	16:27:22	UTC	MA..	FILE	File stat	Content Modifica	-	-	<FILE_LOC>	<FILE_LOC>
03/04/2021	16:32:23	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: users	<FILE_LOC>
03/04/2021	17:33:10	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: J
03/04/2021	17:33:10	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: dialogs	Content: dial
03/04/2021	17:33:24	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: mes
03/04/2021	17:33:48	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: mes
03/04/2021	17:35:49	UTC	.A..	LOG	Telegram Event	Start Time	-	-	Source: messag	Content: Noi

time	source	event	subject	object
Filter	Filter	Filter	Filter	Filter
2021-03-04 16:27:06	File stat	Content Modification ...	Telegram Process	OS:/<file_location>
2021-03-04 16:27:07	File stat	Content Modification ...	Telegram Process	OS:/<file_location>
2021-03-04 16:27:13	File stat	Content Modification ...	Telegram Process	OS:/<file_location>
2021-03-04 16:27:22	File stat	Content Modification ...	Telegram Process	OS:/<file_location>
2021-03-04 16:27:22	File stat	Content Modification ...	Telegram Process	OS:/<file_location>
2021-03-04 16:32:23	Telegram Event	User Status Update		username:USER firstname:...
2021-03-04 17:33:10	Telegram Event	Telegram Message Sent	Telegram	J
2021-03-04 17:33:10	Telegram Event	Chat Last Update		dialog unread_count:0 inbox_max:40
2021-03-04 17:33:24	Telegram Event	Telegram Message Sent	Contact_1	message_action_history_clear
2021-03-04 17:33:48	Telegram Event	Telegram Message Sent	Contact_1	message_action_history_clear
2021-03-04 17:35:49	Telegram Event	Telegram Message Sent	Contact_2	Noise_6Pj0GUscq4seA

enhanced.csv_timesketch 3

3 events (0.008s)

Datetime (UTC)	☐	Message
2021-03-05T01:28:56	☐	Telegram Message Sent: 'messag
2021-03-05T01:30:13	☐	Telegram Message Sent: 'messag
2021-03-05T01:31:21	☐	Telegram Message Sent: 'Noise_

Figure 5.1: The forensic timeline generation processes.

Chabot et al.'s [28] methodology. It revolves around the creation of a knowledge representation model as derived from multiple forensic sources. It presents a canonical semantic view of the combined sources upon which forensic timeline (or other) analysis can be conducted. This model is populated with scenario events derived from forensic footprints and the raw forensic artefacts collected from different forensic sources. These events are associated with subjects that participate or are affected by the events and the objects acted upon by subjects. Events can then be correlated based on common subjects, objects, temporal relations, or expert rule sets. Event correlation starts with a seed event, an alert the victim raises that investigators consider suspicious due to its unusual nature or pinpointed by the user as not the result of intended device usage. Relations established by this process correspond to a relation of composition or causation.

The first three steps in Figure 5.1 consist of forensic artefact extraction. *JIT-MF_Logs* comprise unique *Evidence_object*s timely dumped from memory, readily carved and parsed. All sources are decoded and merged as a Plaso [56] super timeline using the `psteal` utility, and for which *JIT-MF_Logs* Plaso parser was developed that processes readily-parsed *JIT-MF_Logs* into a single JSON file. A loader utility was developed for Step 4 that traverses the super timeline and populates the knowledge model implemented as an `SQLite` database table. The events in this table correspond to messaging events of some form, depending on the forensic source. For example, JIT-MF Drivers and messaging backups can pinpoint events at the finest possible level of granularity, indicating whether a specific messaging app event is of type send or receive, the recipient/sender. Other sources, such as the file system source (`file stat`), can only provide a coarser level of events related to the reading/writing of app-specific messaging database files on the device. A flat storage model suffices for this experiment, with events considered atomic and their associated subjects and objects corresponding to message recipients and content. Step 5 takes alerts of suspicious activity as input. Alert information associated with some seed event is converted into SQL queries that encode the required subject/object/temporal/event type correlations. The query then outputs those events associated with the initial alert. This event sequence provides the timeline for the incident in question, and for which (step 6), `Timesketch` [53] was used for the visualisation of the said timeline.

5.3.2 Experiment setup

Each case study assumes a high-profile target victim ("John"), whose Android mobile device stores confidential data. John uses multiple messaging apps (SMS and IM) that have been forensically enhanced with JIT-MF due to the potentially heightened threats that his device may face. John receives an email to download a free version of an app that he currently pays for on his mobile device. He downloads it and becomes a victim of a long-term stealthy attack.

Setup. Pushbullet (v18.4.0), Telegram (v5.12.0), and Signal (v5.4.12)[9] are popular SMS and IM apps, respectively, used as targeted apps for messaging app hijack in these case studies. The six case studies comprise messaging hijack scenarios involving spying and crime-proxying by hijacking the functionality of the three aforementioned targeted benign apps. For the case studies involving Telegram and Signal, these attacks are carried out using the Android Metasploit attack suite, whereas, for Pushbullet these attacks are executed through Selenium assuming the attacker procured the victim's credentials.

All possible third-party application logs likely to contain evidence of hijacked attack steps were collected for each app. Yet preliminary analysis showed that for the apps involved (to varying degrees): i) the availability of these logs is improbable; and ii) the content of these logs is subject to tampering as with any other trace deletion techniques.

As a result, JIT-MF Drivers were developed using the *Trigger_point*s and *Evidence_object*s shown in Table 5.6, selected for the six case studies involved in this experiment. Similarly to the setup described in the previous section, app-specific *Evidence_object*s were selected from the app layer since it is most likely that these types of objects will allow JIT-MF to generate *JIT-MF_Logs* that contribute to the most accurate forensic timeline. The selection of *Evidence_object*s was unique to each app, based on analysis of individual app internals. The app-specific *Evidence_object*s for Telegram and Pushbullet remain the same as from the previous experiment. Signal is also an open-source app; therefore, the *Evidence_object* selection process followed that explained for Telegram in the previous section. Since the case studies involve messaging hijack attacks, each app's unique object is the *Message* object representative. *Trigger_point*s were selected from the native layer (with native-level *Trigger_predicate*()) and API layer, which produce the most accurate *JIT-MF_Logs* while imposing the least app invasive and development effort in terms of stack layer comprehension for *Trigger_point* selection (as shown in the results of the previous experiment Section 5.2). From preliminary usage of the apps considered, it was noted that the apps either store/load messages from a local database on the phone or read/send/synced messages over the network. Therefore related system calls (e.g. `send`, `write`, etc.) and API network-related function calls were used as *Trigger_point*s and are listed in each of the case studies below. The drivers used are Listing A.1 - Listing A.6 found in Appendix A.

[9] `https://signal.org/en/`

Table 5.6: *Trigger_point*s and *Evidence_object*s selected.

Case Study	Attack Scenario	*Trigger_point* selection		*Evidence_object* selection	
		Stack layer	*Trigger_point*	Stack layer	*Evidence_object*
A	Telegram Crime-Proxy	Native	`send()`	App	`org.telegram.messenger.MessageObject`
B	Signal Crime-Proxy	Native	`write()`	App	`org.thoughtcrime.securesms.conversation.ConversationMessage`
C	Pushbullet Crime-Proxy	Native	`write()`	App	`org.json.JSONObject`
D	Telegram Spying	Native	`recv()`	App	`org.telegram.messenger.MessageObject`
E	Signal Spying	Native	`open()`	App	`org.thoughtcrime.securesms.conversation.ConversationMessage`
F	Pushbullet Spying	API	`android.content.Intent$1.createFromParcel()`	App	`org.json.JSONObject`

A rooted Google Pixel 3XL developer phone running Android 10 emulator was used in this experiment. This enabled ease of automation and ensured all possible forensic sources were collected from the device (even those found in internal storage) to benefit the baseline timeline generation. Table 5.7 lists the properties of the state-of-the-art forensic sources considered, their method of collection and required parsers for populating the Plaso super timeline. These sources comprised individual app databases, app-specific files including write-ahead log files, system logs and *JIT-MF_Logs*. The write-ahead log files (`cache4.db-wal` files) serve as a cache for maintaining an app's database integrity. They typically contain partial app events which have yet to be inserted inside the app's database and are subject to frequent log rotation. Apart from the JIT-MF Plaso parser, additional parsers for the app-specific databases (including write-ahead-log database) and `logcat` were developed.

Case study setup. In each of the case studies: i) the emulator is started; ii) the targeted app is enhanced with JIT-MF Driver and Driver Runtime; iii) legitimate traffic (noise) is constructed iv) a messaging hijack attack comprising of attack steps is simulated within a controlled setup; v) all available forensic sources of evidence are collected (those typically collected and *JIT-MF_Logs*); and vi) timelines are produced based on a knowledge model. As a result of these steps, the following timelines are generated:

- A *Ground Truth Timeline* generated by logging the attack steps of the executed hijack attack.
- *Baseline Timelines* generated by querying a knowledge model made up of state-of-the-art forensic sources, and
- *JIT-MF Timelines* generated by querying a knowledge model made up of both baseline sources and *JIT-MF_Logs*.

Table 5.7: Forensic sources and parsers.

Case study	**Location on device**	**Source type**	**Contents**	**Collection & Decoding**	**Requires rooting**	**Plaso parsers**
A,D	`/data/org.telegram.messenger/../cache4.db`	Baseline	Telegram database	`adb pull`, Teleparser	Yes	Teleparser parser
A,D	`/data/org.telegram.messenger/../cache4.db-wal`	Baseline	Latest changes to Telegram's database	`adb pull`, Walitean	Yes	Walitean parser
B,E	`/<removable_storage>/.../signal.backup`	Baseline	Signal backup database	Signal DB decryptor	Yes	Signal database parser
D,F	`/data/data/com.android.providers.telephony/.../mmssms.db`	Baseline	SMS database	`adb pull`	Yes	`mmssms.db` Plaso parser
D,F	`/data/data/com.pushbullet.android/.../pushes.db`	Baseline	Pushbullet message database	`adb pull`	Yes	Pushbullet parser
A-F	`/data/<app_pkg_name>/*`	Baseline	App specific files, cache files	`adb pull`	Yes	File stat Plaso parser
A-F	`/<removable_storage>/<app_pkg_name>`	Baseline	Media files	`adb pull`	No	File stat Plaso parser
A-F	`logcat`	Baseline	System logs	`adb logcat`	No	Logcat parser
A-F	`/<removable_storage>/jitmflogs`	*JIT-MF_Logs*	*Evidence_object*s dumped from memory	`adb pull`	No	JIT-MF parser

While there is only one ground truth timeline, multiple JIT-MF and baseline timelines can be created per case study depending on the different seed event correlations. These timelines are populated with events outputted from a query run on the knowledge model that starts from a seed event.

Background noise was generated, taking into account the app's functionality. That is, the messaging app was loaded and messages were exchanged via the targeted app involved in the messaging hijack attack simulation. Each attack simulation comprised background noise and three malicious events (three crime-proxy events or three spying events). Some of the attacks in these case studies target victim apps that use rate-limited API calls to a server backend, allowing only 150 consecutive calls from the same device. Since each attack comprised three such events per case study, and the API call limit is 150, each attack simulation was executed fifty times — each time obtaining the timelines above. For each case study, the specifics of the simulated attack are described along with the complete experiment setup (including background noise and attack parameters) and the investigation process (including the seed event and matching criteria).

Timeline comparison. The *JIT-MF Timeline* and *Baseline Timeline* are compared to the *Ground Truth Timeline* based on: i) completeness of timeline, i.e. lack of missing events; ii) accuracy of the timelines concerning the sequence in which the events happened and the difference between the recorded time of an event in the ground truth timeline and the JIT-MF timeline. Preliminary runs showed that baseline forensic sources could provide different metadata depending on the benign app hijacked. Therefore, the matching criteria for a matched event between the generated and ground truth timelines are adjusted in the case studies to benefit from the evidence typically found in baseline forensic sources.

A: Telegram Crime-Proxy

Accessibility attack. An accessibility attack targets John's Telegram app and is used by an attacker to send messages to a co-conspirator using the username "Alice" on Telegram. The attacker misuses the victim's Telegram app to send messages to "Alice" and instantly deletes them.

Setup. John uses his Telegram app regularly to communicate with his family and friends. He sends six Telegram messages to his relatives before entering a meeting, then goes silent. The attacker notices the decrease in Telegram activity and uses this time to communicate with "Alice" three times. He waits ten to twenty seconds (randomly generated using `rand`) every time before messaging "Alice". The attacker tries to execute the attack as quickly as possible to retain stealth but gives an allowance of ten seconds to allow for any delays within the app. John continues using Telegram thereafter and sends six messages to his friend. John's messages take this form: $Noise\_ < Random10 - 100 - letters >$ whereas those sent by the attacker are similar to *Sending_Attack_#Iteration*.

Investigation. John notices a new chat on his phone with the username "Alice" with no messages. He brushes it off but is contacted later that week by investigators who told him that his phone was used to send messages containing the specific keywords. He takes his phone to be examined. His phone is already equipped with a JIT-MF Driver as shown in Listing A.1.

This attack step involves the sending of a message over the network. Therefore the selected *Trigger_point* is the `send` system call, and the *Evidence_object* is the Telegram message itself.

The seed event is generated based on the alert flagged, which gives the investigators three possible starting points to use when formulating the queries to be executed on the different knowledge models.
Seed Event: *Subject: Alice, Object: *specific keywords*, Event type: Message Sent, Time: last seven days*
Matching criteria: The criteria for an event in the baseline or enhanced timelines to match the ground truth timeline is the presence of the specific message content that was sent within the event.

B: Signal Crime-Proxy

Accessibility attack. An accessibility attack targets John's Signal app and is used by an attacker to send messages to a co-conspirator using the username "Alice" on Signal. The attacker misuses the victim's Signal app to send messages to "Alice" and instantly deletes them.

Setup. This case study is identical to the one described in the previous section.

Investigation. John's phone is already equipped with a JIT-MF Driver as shown in Listing A.2. Like the previous case study, the *Evidence_object* is the Signal message itself. The `send` system call is not called when sending a message. Thererfore, the `write` system call is used as a *Trigger_point* instead, which writes to the local database and over the network.

Seed Event: *Subject: Alice, Object: *specific keywords*, Event type: Message Sent, Time: last seven days*
Matching criteria: An event stating that a message was sent to Alice's number.

C: Pushbullet Crime-Proxy

Accessibility attack. John's Facebook credentials are stolen by an attacker using a phishing accessibility attack akin to Eventbot [138]. The attacker uses the stolen credentials to proxy SMSs, through John's Pushbullet app, from his web browser.

Setup. John does not use SMS functionality on his phone but is aware that he receives many advertisement messages. John receives six ad messages before entering a meeting. The attacker notices the decrease in activity and uses this time to communicate with "Alice" three times. He waits ten to twenty seconds (randomly generated using `rand`), then opens his browser and sends three messages to "Alice". Messages received by John take this form: $Noise\_ < Random10 - 100 - letters >$ whereas those sent by the attacker are similar to this: *Sending_Attack_#Iteration*.

Investigation. John receives a hefty bill from his telephony provider at the end of the month, attributing most of the cost to message sending. He notices a new number not on his contact list and initiates a forensic investigation. His phone is already equipped with a JIT-MF Driver as shown in Listing A.3.

Pushbullet stores message objects in JSON structures. A `write` system call *Trigger_point* occurs when a message is sent, at which point the process memory contains the message sent, stored in JSON.
Matching criteria: A message sent to the suspicious number.

D: Telegram Spying

Accessibility attack. An accessibility attack targets John's Telegram app and is used by an attacker to intercept messages sent to the username "CEO" (John's boss - with whom confidential data is shared). The attacker misuses John's Telegram app to grab messages exchanged with "CEO" and Telegram.

Setup. John regularly uses his Telegram app to communicate with his CEO. John sends messages to his CEO multiple times during the day but goes silent during three meetings. The attacker notices the decrease in Telegram activity and decides to use this time to spy on John's correspondence with his CEO. He waits ten to twenty seconds (randomly generated using `rand`), then opens Telegram, loads the "CEO" chat, intercepts the messages loaded on the screen, then closes the app quickly. Messages sent by John take this form: $Confidential\_ < Random10 - 100 - letters >$.

Investigation. John's phone is already equipped with a JIT-MF Driver as shown in Listing A.4. In the case of spying, one of the attack steps involves reading a message. Therefore, the *Evidence_object* is the message itself. Since Telegram is a cloud-based app, some messages are stored on the device, and others are loaded and received from cloud storage over the network. Therefore the selected *Trigger_point* is the `recv` system call.
Seed Event: *Subject: CEO, Object: *confidential message*, Event type: Message Read/Loaded/Chat activity, Time: date of message sent/received*
Matching criteria: An event type indicating chat activity, loading, or reading of "CEO" messages. The message object itself does not correspond directly to an attack step. The message object in memory does not contain metadata about whether it was read but rather that it was either sent or received at some point. JIT-MF forensic sources identify a *chat interception event* instead as multiple message objects exchanged with the same contact, all having been dumped at the same timestamp. Furthermore, the timestamp of these events must occur in the database after the sending time to avoid including data related to when the message was initially sent or received.

E: Signal Spying

Accessibility attack. An accessibility attack targets John's Signal app and is used by an attacker to intercept messages sent to the username "CEO". The attacker misuses John's Signal app to open a confidential chat with the username "CEO" and grabs the messages that appear on the screen. Finally, the attacker closes Signal.

Setup. This case study is identical to the previous one.

Investigation. John's phone already has a JIT-MF Driver, as shown in Listing A.5. Like the previous case study, the *Evidence_object* is the intercepted Signal message. Signal is not a cloud-based app and uses solely on-device storage. Therefore the `open` system call, used to open the database file from which messages are loaded to be read, was selected as a *Trigger_point*.
Seed Event: *Subject: CEO, Object: *confidential message*, Event type: Message Read/Loaded/Chat activity, Time: date of message sent/received*
Matching criteria: As previous case study.

F: Pushbullet Spying

Accessibility attack. John's Facebook credentials are stolen by an attacker using a phishing accessibility attack. The attacker now has access to any messages sent or received by John through a syncing event on John's phone.

Setup. John regularly uses his SMS app to communicate with his CEO. John sends messages to his CEO multiple times during the day but goes silent during three meetings. Unbeknownst to him, the attacker immediately intercepts John's ongoing SMS activity.

Investigation. John's phone is already equipped with a JIT-MF Driver as shown in Listing A.6. Unlike Telegram and Signal, Pushbullet spawns several sub-processes to sync activity generated on the device with that stored in the cloud. While in Case Study C the attack involves a level of interaction with the device (since the SMS has to be sent from the device after receiving an instruction from the browser), in this case, any message sent or received is assumed to have been intercepted automatically. The *Trigger_point* selected is one of the Android API calls used by the Pushbullet to sync sent/received messages via Firebase. The only *Evidence_object*, related to an attack step, that can be retrieved from memory for this case study, is a JSON object containing "push" event metadata which indicates message content has been synced and can be remotely read.
Seed Event: *Subject: CEO, Object: *confidential message*, Event type: Message Read/Loaded/Chat activity, Time: date of message sent/received*
Matching criteria: As previous case study.

5.3.3 Results

Table 5.8 compares the generated JIT-MF timelines and Baseline timelines, per seed event correlation, to the ground truth timeline. The generated timelines included events unrelated to the attack steps (noise generated when sending and receiving legitimate messages); therefore, *precision* and *recall* were used. Precision is a value between 0 and 1, which denotes the average relevant captured events. The higher the value, the larger the portion that attack steps make up the timeline, i.e. little noise was present. Recall denotes how many of the executed attack steps were uncovered. Similarly, the higher the value between 0 and 1, the more attack steps in the ground truth timeline were captured. Timeline difference in the captured attack events from the ground truth timeline was calculated using *Jaccard dissimilarity* on the set of attack events uncovered by the generated timelines. In this case, the higher the value between 0 and 1, the more dissimilar (undesirable) the generated timeline is to the ground truth.

Table 5.8: Forensic timeline comparisons.

Case study	**Seed event - Correlation**	**Baseline**			**JIT-MF Timeline**		
		Recall	Precision	Timeline difference	Recall	Precision	Timeline difference
A	Subject	0	-	1	0.98	1	0.02
	Object	1	0.66	0	1	0.66	0
	Event Type	1	0.01	0	1	0.01	0
B	Subject	1	0.07	0	1	0.06	0
	Object	0	-	1	0.87	1	0.13
	Event Type	1	0.11	0	1	0.07	0
C	Subject	1	1	0	1	1	0
	Event Type	1	0.23	0	1	0.23	0
D	Subject	0	-	1	0.49	0.46	0.51
	Object	0	-	1	0.49	0.45	0.51
	Event Type	0	-	1	0.49	0.45	0.51
E	Subject	0.99	0.97	0.01	0.99	0.21	0.01
	Object	0	-	1	0.58	0.23	0.42
	Event Type	0.13	0.01	0.87	0.63	0.02	0.37
F	Subject	0	0	1	0	0	1
	Object	0	0	1	0	0	1
	Event Type	0	-	1	0.02	1	0.98

- denotes no events were captured, so precision could not be calculated.

Primary contributors to timeline similarity. The timeline difference values in the table show that overall JIT-MF timelines are *at least* as similar to the ground truth as baseline timelines. While the dissimilarity for the baseline timelines varies substantially *within a single case study*, depending on the seed event - correlation, this is not the case for JIT-MF timelines whose distance from ground truth remains roughly the same. Since JIT-MF forensic sources include crucial evidence metadata (message content, recipient, date ...), the chosen seed event correlation has little to no effect on the output timeline. In contrast, evidence from baseline sources is not as rich, with correlation becoming a critical factor affecting the resulting timelines. Yet even in scenarios where JIT-MF timelines are equivalent to the baseline in event sequences, the metadata available in JIT-MF forensic sources can provide the investigator with richer timelines through more informative events. For case study C where the timeline similarity of the JIT-MF timeline was equal to the baseline and this was consistent across different seed events, this can be attributed to the fact that the Pushbullet attack was not as stealthy and did not include removing forensic footprints (as the app does not expose this functionality). This shows that the impact of *JIT-MF_Logs* on the forensic timeline is more significant when the attack is stealthier.

The table also shows that JIT-MF timelines are still more similar to the ground truth in the case of spying (case studies D-F) when compared to the baseline sources, which do not include evidence pointing to message reading or browsing chat activity.

A point of concern is that in the instances when the JIT-MF timeline difference is low; that is, the JIT-MF timeline is close to the ground truth, the precision value is low, indicating that a lot of *Evidence_object*s are present in the timeline that are not part of the attack steps. While collected *Evidence_object*s mean that evidence of attack steps is present, this also shows that further post-processing effort is required to address precision in forensic timeline generation of attack steps. This is addressed in a separate evaluation presented in Section 6.2.

Primary contributors to timeline dissimilarity. JIT-MF timelines were most dissimilar from the ground truth in the last case study F. In Pushbullet's case, the app is hijacked differently than Telegram and Signal. For the Telegram and Signal attack scenarios, the malware is present on the device, taking over the hijacked app directly. Whereas with Pushbullet, the attack is leveraging compromised credentials from a remote device or workstation, and no malware may be present on the victim's device. Yet, JIT-MF Drivers are attack vector agnostic; therefore, this should not affect *JIT-MF_Logs* contents. The factors that could have contributed to this include: i) many of the app's functionality was delegated to a sub-process that was not instrumented; ii) the *Evidence_object* defined in the JIT-MF Driver was coarser-grained (a JSON object containing "push" event that synced changes) that did not include the necessary metadata for correlating spying events. These limitations in the JIT-MF's driver implementation contributed to a JIT-MF timeline whose gain on the baseline timeline was minimal regarding ground truth timeline similarity.

Modifications needed to be made to the JIT-MF Driver template to mitigate issues related to an app having delegated functionality to multiple processes. Listing 4.1 *line 3* indicates how JIT-MF Drivers may be fine-tuned, so specific app processes are instrumented rather than automatically instrumenting the main app process. Yet, this may still be insufficient to address the other reason for dissimilarity in the Pushbullet spying case study.

Furthermore, while JIT-MF timelines are more similar to the ground truth timeline than baseline timelines in case studies D-F involving spying, they are less similar to the ground truth timelines when compared to JIT-MF timelines obtained for the crime proxy case studies A-C. The difference between these sets of case studies is that in crime proxy scenarios, the *Evidence_object* and *Trigger_point* defined in the JIT-MF Driver are tightly linked. The results are *JIT-MF_Logs* containing message objects with metadata that can be linked to the hijacked app functionality of interest. In spying scenarios, events are coarse-grained (an indication of a chat being intercepted/synced rather than an individual message). Moreover, these events are not tightly coupled with *Trigger_point*s since key objects in memory are either absent or do not contain indicative metadata of the ongoing event. This shows that the selection of *Evidence_object*s and *Trigger_point*s merits further effort in different attack scenarios to ensure that they reflect app functionality of interest and that they can be linked to the hijacked attack step that the JIT-MF Driver was developed to log.

JIT-MF timeline sequence accuracy. When performing order-sensitive comparisons using *Kendall Tau coefficient*, one can conclude that the sequence of captured events in JIT-MF timelines (containing only ground truth events) is always identical to that in the ground truth timeline, i.e. a coefficient of 1 is observed in all cases. Additionally, the standard deviation between the time of the events logged in the ground truth timelines and that logged in JIT-MF timelines is at most 62s. While any additional cost to complete a typical app function diminishes the app's performance, the delay occurs during message sending and receiving without affecting the user interface; therefore, the lag is not noticeable.

Performance overheads. Since Pushbullet offers remote SMS-on-PC functionality, performance overhead was calculated based on the increase in turnaround time. With Telegram and Signal, these apps are generally used through the phone's UI; therefore, performance overhead was measured in Janky frames,[10] an indicator of non-smooth user interactions with GUI apps. With JIT-MF Drivers enabled, only an average increase of 0.5s was registered in Pushbullet turnaround times for SMS operations, as observed from the web browser's Javascript console. Telegram and Signal had an average increase of 1.59% and 3.53% of Janky frames, respectively, with JIT-MF Drivers enabled; the performance penalty overall was less than 3.53%.

While Janky frames indicate how smooth the app's UI is while running, this does not factor in performance issues such as the app crashing, a few instances of which were observed during experimentation with Telegram and Signal apps. Furthermore, while the experiment ran relatively short, logs generated by JIT-MF and stored on the device accumulate until they are collected and removed by a potential investigator. Therefore, if the magnitude of these logs increases exponentially over time, the device would be

[10] `https://developer.android.com/topic/performance/vitals/render`

rendered unusable. Both of these issues influence the feasibility of JIT-MF in a realistic setting and require further experimentation that contributes to maintaining app stability. An experiment addressing this concern is presented in Section 5.5.

5.4 Evidence object accuracy in lower layers of the stack

Results from the previous section show that app-specific artefacts within *JIT-MF_Logs* can contribute towards a more accurate forensic timeline, concerning baseline sources. Specifically, forensic timelines generated using *Trigger_point*s from lower levels in the stack, produce accurate forensic timelines for IM crime-proxy messaging hijack attacks. The selection of *Trigger_point*s from lower levels in the stack assists towards achieving a minimally invasive approach that requires less development effort due to *Trigger_point* instructions selected from layers of the stack that are more openly documented than app-specific instructions. Yet so far, the selection of *Evidence_object*s has been made from the app layer. This approach the most obvious way to associate memory artefacts with an application functionality of interest; and therefore guarantees the accuracy of the app-specific artefacts collected and which contribute to forensic timeline generation. Therefore, an app-invasive approach is still required to comprehend app layer internals and establish *Evidence_object*s. Thus imposing infeasible JIT-MF Driver development, due to required knowledge of app internals to determine app-specific *Evidence_object*s associated with app functionality of interest.

The experiment described in this section aims to determine how JIT-MF Drivers can be generalised further without compromising timeline accuracy. It evaluates the accuracy of *JIT-MF_Logs* when using JIT-MF Drivers whose both *Trigger_point*s and *Evidence_object*s are selected from the API layer of the Android technology stack. While *Trigger_point*s from this layer are expected to be more app-specific than those selected from the native layer due to app-specific usage, the publicly-available documentation and widespread usage of APIs across apps show the potential for *Trigger_point*s positioned in this layer. Furthermore, given the performance overheads resulting from the previous experiment utilising *Trigger_point*s from the native layer and initial indications from performance results obtained in the first experiment (Table 5.5), it is possible that *Trigger_point*s at this layer are likely to cause the least concern regarding storage and execution overheads.

5.4.1 API-based JIT-MF Driver development

The API layer in the Android technology stack is generally more stable and widespread across different applications and versions, allowing JIT-MF Drivers to be developed

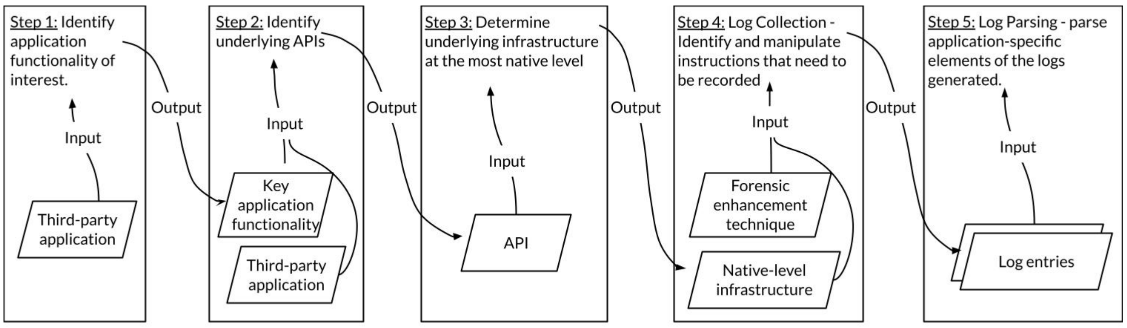

Step 1: Identify application functionality of interest.
Input
Third-party application
Output
Step 2: Identify underlying APIs
Input
Key application functionality
Third-party application
Output
Step 3: Determine underlying infrastructure at the most native level
Input
API
Output
Step 4: Log Collection - Identify and manipulate instructions that need to be recorded
Input
Forensic enhancement technique
Native-level infrastructure
Output
Step 5: Log Parsing - parse application-specific elements of the logs generated.
Input
Log entries

Figure 5.2: Proposed process for creating and deploying API-based JIT-MF Drivers.

in a minimally-invasive approach. In return, JIT-MF Drivers with *Trigger_point*s and *Evidence_object*s selected from the API layer remain usable across different applications and versions using the same API for app functionality of interest (unlike application-specific JIT-MF Drivers, which need to be developed anew). Figure 5.2 presents the process for creating *API-based JIT-MF Drivers* as part of this new proposed approach.

Step 1: The first step in developing an API-based JIT-MF Driver requires that, given a third-party application, a JIT-MF Driver developer must first identify the scope of the JIT-MF Driver by determining which key application functionality requires deeper visibility (application functionality of interest) and hence needs to be logged. In the case of stealthy messaging attacks, for instance, key application functionality constitutes messaging events that an attacker could hijack.

Step 2: Application developers use readily-available infrastructure (such as services, libraries and APIs) to develop key functionality commonly carried out among applications. Given a third-party application, the JIT-MF Driver developer must determine the underlying APIs and their instructions used to perform the identified key application functionality; for example, storage/database libraries that persist messaging events in the application's local database.

Step 3: Applications using the same underlying infrastructure may use custom APIs at higher abstraction levels that better fit a specific application's needs. However, at the native level, these APIs typically use the same native libraries. Therefore the JIT-MF Driver developer must determine the underlying infrastructure at the most native level, which is expected to be consistent among application versions and applications; for instance, the native `sqlite.c` library.

Step 4: Given a forensic enhancement technique and the underlying infrastructure identified, publicly available documentation of the interface exposed by the infrastructure's API can be used to determine *Evidence_object*s and *Trigger_point*s. In the case of API-based JIT-MF Drivers, the *Trigger_point*s selected comprise infrastructure instructions which include *Evidence_object*s as parameters that comprise evidence of the hijacked attack steps as a result of application functionality of interest being executed. Since the focus is on API instructions, this step involves analysing only a subset of the application codebase that interacts with the API. This is typically publicly well-documented through the API interface and can be *common* for applications that use the same underlying infrastructure.

Step 5: An infrastructure codebase is expected to expose the same interface across the different applications in which it is used. However, applications may make application-specific use of the interface and the instructions carried out by the infrastructure. Therefore, the final step is expected to be unique to each application and involve application-specific parsing of the generated and collected log entries. This means that an element of compiled app code analysis is still required to make sense of the application-specific elements in the log entries, the extent of which is difficult to predict as it requires a large-scale qualitative exploration of the solution. However, it is expected that the definition of the parser will be similar for applications using the same underlying infrastructure. Since infrastructure code is more stable and less likely to change, modifications to the log parser between an application's versions are expected to be minimal. While the parsing element may need revising, the collection element remains functional. Therefore, evidence is collected regardless of whether or not a parser is readily available or a new one needs to be developed.

5.4.2 Common infrastructure adoption

A preliminary analysis is necessary to identify which infrastructure is widely adopted among messaging apps to ensure that an API-based JIT-MF Driver can function across app versions and on multiple apps. Not only does this infrastructure need to be in use by the app, but it also needs to be taking an *active role* when application functionality of interest that needs to be logged is taking place. AppBrain [14] is a service that provides statistics on the Android application's ecosystem, including library adoption by different apps in different categories. In this case, the data provided by AppBrain specifically on Android messaging apps is used per the scope of this evaluation.

AppBrain categorises libraries used in Android applications by tags, depending on the functionality provided by the library. As described in step 2 of Figure 5.2, when developing an API-based JIT-MF Driver, the selected infrastructure must support core app functionality that is within the JIT-MF Driver's scope and requires deeper visibility (Figure 5.2 step 1). Out of the 41 possible categories, *Network* and *Database* libraries are identified as critical infrastructures used by messaging apps to support key messaging functionality. Network functionality allows messages to be sent and received over the network, and Storage allows messages (both sent and received) to be stored and retrieved on the devices where the app is installed.

At the time of this analysis, AppBrain statistics showed that the most widely used network library was Retrofit.[11] In contrast, the most widely used database infrastructure was Android Architecture Components,[12] which at its most native level (see Step 3 Figure 5.2) refers to storage management through an SQLite Database.[13] AppBrain statistics revealed that at the time, 86.62% of communication (messaging) apps used Android Architecture Components, while only 14.6% used Retrofit.

5.4.3 Case study experiment

This experiment was carried out using representative apps from the same category; that is, instant messaging (IM) apps. To this end, the Pushbullet app used in the previous case studies was replaced with the WhatsApp app. Furthermore, given that results from previous experiments showed that further research is required to devise better JIT-MF Drivers for spying scenarios, the experiments within this case study focus on a crime-proxy attack scenario.

Telegram (v8.8.5), Whatsapp (v2.22.17.70) and Signal (v5.44.4) were installed on two Google Pixel 3XL emulators running Android 10 (API 29). JIT-MF was used to enhance the three apps on one of the emulators (Device A) with a JIT-MF Driver. The JIT-MF Drivers from the previous experiment (with *Evidence_object* selected from the App layer) were used to forensically enhance Telegram and Signal. A new JIT-MF Driver was created for WhatsApp using an app-specific *Evidence_object*. A single new SQLite API-based JIT-MF Driver[14] was also developed to be used on all three apps. AndroidViewClient[15]

[11]https://square.github.io/retrofit/
[12]https://www.appbrain.com/stats/libraries
[13]https://developer.android.com/training/data-storage/sqlite
[14]https://gitlab.com/bellj/infrastructure_based_agents
[15]https://github.com/dtmilano/AndroidViewClient

was used to simulate normal messaging traffic, whereby 20 messages were sent, and 20 messages were received by Device A. Each message was formulated as follows: $Normal\_Message\_ < \#msgnumber >$.

API-based JIT-MF Driver accuracy comparison. Resulting *JIT-MF_Logs* produced by the drivers are expected to include any messages sent and received by the enhanced app. The accuracy of an SQLite API-based JIT-MF Driver is compared to JIT-MF Drivers with app-specific *Evidence_object*s, based on the log entries found in *JIT-MF_Logs* using the different JIT-MF Drivers.

API-based JIT-MF Driver development. Following the steps shown in Figure 5.2, the SQLite API-based driver was developed, as follows. Message send functionality was identified as the key application functionality of interest the SQLite API-based JIT-MF Driver must log. Log entries produced by JIT-MF are invoked at *Trigger_point*s containing *Evidence_object*s. In the case of an SQLite API-based JIT-MF Driver, *Trigger_point*s were defined as the functions exposed by the `sqlite` C interface[16] and *Evidence_object*s are defined as the parameter within these functions that contains the SQL statements executed.

Log collection. The publicly available documentation for SQLite indicates that prepared statements allow applications to execute all SQL statements. Furthermore, the second parameter of any prepared statement contains the SQL query to be executed. Subsequent bind functions (`BIND_INT,BIND_TEXT,BIND_BLOB`) are used to populate the parameterised values of the query, depending on a query ID.[17] Therefore log entries produced by SQLite API-based JIT-MF Driver should consist of SQL statements that were executed to populate messages in the messaging app local database.

Log parsing. A log parser was required to parse the resulting *JIT-MF_Logs*. While the SQLite API-based JIT-MF Driver was the same for all three apps, the log entries contained app-specific knowledge requiring unique parsing logic per app. For instance, not all apps use the same SQL statements to populate their respective local databases. However, SQLite databases include a schema that describes the tables within a database. Therefore the developed parser could use this information to carve out the necessary portions from the log entries.

[16]`https://www.sqlite.org/c3ref/funclist.html`
[17]`https://www.sqlite.org/c3ref/stmt.html`

```
1 INSERT INTO message_ftsv2(fts_jid,docid,content,fts_namespace) VALUES (0 b,null,normal_message_1
      ,0 b)
2 REPLACE INTO messages_v2 VALUES(2328, 1679923803, 2, 0, 1662483779, n8\"QY[!d\"QY[!dC}
      cNormal_message_1 , 0, 0, 18446744073709552000, NULL, 0, 0, 0, undefined, 0, 0, 0, undefined
      )
3 INSERT INTO sms(thread_id,subscription_id,address,protocol,expires_in,server_guid,date_sent,body
      ,date,read,type,unidentified,date_server,reply_path_present,service_center,address_device_id
      ) VALUES (3,18446...0,3, 31337,0,d500 3b3d-ce8e-41cf-ba67-1cf592fa81c2,1662485982439,
      Normal_message_1Âğ,1662485983160,0,10485780, 1,1662485974012,1,GCM,1)
```

Listing 5.1: Unparsed log entries generated by infrastructure-based JIT-MF Driver for WhatsApp (1), Telegram (2) and Signal (3) respectively

```
1 {"time": "1662481636", "event": "WhatsApp Message Sent", "trigger_point(s)":
      "sqlite3_clear_bindings|sqlite3_prepare_v2|sqlite3_prepare16_v2|sqlite3_bind_int|
      sqlite3_bind_int64|sqlite3_bind_text|sqlite3_bind_text16|sqlite3_bind_blob|sqlite3_finalize"
      , "object": {"date": "", "message_id": "","text": "normal_message_1", "to_id": "", "to_name"
      : "", "to_phone": "", "from_id": "", "from_name": "", "from_phone": ""}}
2 {"time": "1662483789", "event": "Message Sent", "trigger_point(s)": "sqlite3_clear_bindings|
      sqlite3_prepare_v2|sqlite3_prepare16_v2|sqlite3_bind_int|sqlite3_bind_int64|
      sqlite3_bind_text|sqlite3_bind_text16|sqlite3_bind_blob|sqlite3_finalize", "object": {
      "message_number": "2328", "date": "1662483779", "text": "Normal_message_1 ", "type":
      "received", "to_id": "5181266731", "to_name": "target_phone;;;", "to_phone": "35699626972",
      "from_id": "1679923803",  "from_name": "contact_phone;;;", "from_phone": "35679247196"}}
3 {"time": "1662485983", "event": "Message Sent", "trigger_point(s)": "sqlite3_clear_bindings|
      sqlite3_prepare_v2|sqlite3_prepare16_v2|sqlite3_bind_int|sqlite3_bind_int64|
      sqlite3_bind_text|sqlite3_bind_text16|sqlite3_bind_blob|sqlite3_finalize", "object": {"date"
      : "1662485983160", "text": "Normal_message_1", "type": "received", , "to_id": "_", "to_name"
      : "target_phone", "to_phone": "+35699626972", "from_id": "3", "from_name": "contact_phone",
      "from_phone": "+35679247196"}}
```

Listing 5.2: JIT-MF log entry sample generated while using WhatsApp, Telegram and Signal, produced through an SQLite infrastructure-based JIT-MF Driver.

Listing 5.1 shows three unparsed log entries generated by an SQLite infrastructure-based driver, while sending messages from WhatsApp, Telegram and Signal, respectively. Each entry comprises an `INSERT | REPLACE` statement, signifying a new message that will be inserted in the local database. App-specific parsers were developed which could: i) identify the names of the `message` table unique to each app (`message_ftsv2`, `messages_v2` and `sms`), and ii) map the appropriate *field values* with the key application functionality metadata; for instance, the second field value of a **`REPLACE INTO`** `messages_v2` table generated when using Telegram reflects the time at which a message was sent. The respective parsed log entries are shown in Listing 5.2.

Results. Publicly-available documentation allowed the development of an SQLite API-based JIT-MF Driver minimising substantially the app-specific compiled code analysis. The same SQLite API-based JIT-MF Driver was effective on all three apps; however, app-specific log parsing was required per app as expected due to app-specific usage of the API, after *JIT-MF_log* collection.

Output generated by the SQLite API-based JIT-MF Driver was compared with that produced by the application-based JIT-MF Drivers. The final column in Table 5.9 shows

Table 5.9: The table shows the maximum lines of application code (LoC) that need to be analysed to develop an app-specific JIT-MF Driver and an SQLite API-based JIT-MF Driver, along with the percentage of log entries retrieved by an SQLite API-based JIT-MF Driver, when compared with log entries generated using an app-specific JIT-MF Driver.

Application	**Maximum LoC analysed for app-specific JIT-MF Driver**	**Maximum LoC analysed for SQLite API-based JIT-MF Driver**	**% of log entries retrieved by SQLite API-based JIT-MF Driver**
WhatsApp	1,515,334	36,778	100
Telegram	1,025,467	50,964	100
Signal	1,552,171	61,496	100

that the SQLite API-based JIT-MF Driver produced the same number of log entries containing the same text metadata, as each app-specific driver. While *text* metadata in log entries generated by the SQLite API-based JIT-MF Driver (Listing 5.2) and application-based JIT-MF Drivers (Listing 4.4) is the same, closer inspection of the two sample log entry outputs generated, shows some difference in the other *Evidence_object* metadata generated. *Evidence_object* metadata generated by SQLite API-based JIT-MF Driver translates roughly to the same metadata obtained by the application-based JIT-MF Drivers in the case of Telegram and Signal. With WhatsApp, however, *Evidence_object* log entries are missing the recipient value. Unlike Telegram and Signal, WhatsApp is closed-source. Therefore, parsing the generated logs was not as straightforward and possibly required further in-depth compiled code analysis efforts to parse metadata within SQLite API-based log entries fully. That said, given that the log entries generated by the JIT-MF Driver contain all the *Evidence_object* involved in interactions with the database, additional code comprehension effort can be carried out at a later stage to parse the generated *JIT-MF_Logs*.

App-specific *Evidence_object*s as defined in JIT-MF Drivers are selected based on comprehension and analysis of the application codebase, which calls for compiled and possibly obfuscated app-specific code analysis. Lines of code (LoC) are used as a metric to highlight the effort needed to analyse compiled application code when developing app-specific and SQLite API-based JIT-MF Drivers. The second and third columns of Table 5.9 show the *maximum* LoC that need to be analysed using both approaches (app-specific and SQLite API-based JIT-MF Driver) to select the relevant *Evidence_object* and thus develop a JIT-MF Driver for each app. For app-specific JIT-MF Drivers, LoC for each app reflects the lines of code in the respective unpacked `java` source files. For the SQLite API-based JIT-MF Driver, the LoC reflects the lines of code of unpacked `java` source files that interact with the SQLite API. In all three cases, the app codebase that needs to be analysed to parse the *JIT-MF_Logs* produced by SQLite JIT-MF Driver is much smaller than each of the applications'. The LoC values in the table portray the worst-case scenario, as analysis efforts can be reduced further in the case of app-specific JIT-MF Driver by using keyword searches (e.g. *Message*) – assuming an unobfuscated codebase – to narrow the search for the *Evidence_object* to a couple of classes. Similarly, public documentation of the infrastructure's interface can outline key exposed functions that can narrow the search even further to the select classes that use those functions.

Regarding performance, no UI degradations were observed during experimentation, indicating no app performance degradation or slow-down. Given initial performance results in Table 5.5 and observation during this experiment, it is possible that API-based JIT-MF Drivers are not only optimal in terms of the minimally invasive approach required to select *Trigger_point*s and *Evidence_object*s but also in terms of performance overheads, thus avoiding the need for sampling, altogether. This, however, would require further experimentation to deduce and may not necessarily generalise across different attack scenarios and APIs.

5.4.4 Prevalence across app versions

The benefits of a generic driver can be measured across apps and across app versions, whose frequent changes in codebases may require JIT-MF Driver developers to update JIT-MF Drivers just as frequently. Results from the preliminary analysis show that storage libraries, specifically managed through SQLite, are the most commonly-used storage library and surpass the usage of network libraries in messaging apps by 68.02%. While results from the case study for WhatsApp, Telegram and Signal, show that an API-Based JIT-MF Driver is effective across the ***latest*** versions of these apps, it remains to be seen whether or not this API can be relied on across previous versions of the apps.

AppBrain does not provide statistics on the previous versions of Android applications. Therefore, a quantitative static analysis approach is used within a qualitative set of applications (comprising WhatsApp, Telegram and Signal) to assess the longevity of an SQLite API-based JIT-MF Driver across application versions.

5.4.4.1 Experiment setup

Previous versions of each app were obtained from August 2017 every six months up until August 2022 (10 past versions per app, 30 apps in total), using APKCombo,[18] a repository for apps and their previous versions. Versions of the apps dating more than

[18]`https://apkcombo.com/`

six months prior could not be installed successfully due to limitations and restrictions presented by each app. Telegram was the only exception which allowed versions from six-month before being run. Due to this limitation, a static check was used to assert SQLite library usage in the unpacked sources of an app's version. This check is carried out based on a signature that encapsulates how a library can be used in an Android app. To use a library, Android apps can either: i) call Android's API wrapper or ii) use a custom implementation that interfaces with the infrastructure using JNI by including a shared object in the APK.

The search aims to (1) find *smali* code in unpacked code that calls functions from the SQLite Android package or (2) find the SQLite shared object in the application library folder of the unpacked app. If the search is successful, the search returns the parent folder name; that is, the app version folder.

```
grep -Priq --include *.smali "Landroid/database/sqlite.*;->"
$app_version/smali*/ |
grep -riq sqlite $app_version/lib/
```

Listing 5.3: Signature for checking the presence of SQLite usage in an APK.

5.4.4.2 Results

Table 5.10 shows the results obtained when the signature in Listing 5.3 is used to check for SQLite usage within the unpacked application versions obtained. These results show that bindings with the common underlying SQLite infrastructure are present across all apps and previous versions from the last five years, even if the interfacing method has changed. Therefore, selecting an infrastructure-based JIT-MF Driver based on SQLite in the case of WhatsApp, Telegram and Signal is likely to remain compatible with upcoming versions of the apps.

Table 5.11 shows the frequency by which newer versions of the apps and SQLite are released. Application-specific JIT-MF Drivers for WhatsApp, Telegram and Signal are based on application codebases that are updated on average every 6 - 15 days. Therefore, a compatibility test would be required to ensure that application-specific logic within the Driver was not altered in the update. On the other hand, updates to the SQLite library are much less frequent and optional, hence not necessarily reflected in the apps that use them. Furthermore, in the case of SQLite, stable interfaces are maintained indefinitely in a backward-compatible way,[19] which means that the JIT-MF Driver relying on an older

[19]https://www.sqlite.org/capi3ref.html

version of SQLite will remain compatible, even if the infrastructure is updated.

Table 5.10: The table shows whether or not the SQLite usage signature was matched in different versions of Signal, Telegram and WhatsApp, since 2017.

Release Date	App	App version	Found SQLite function calls in disassembled smali code (1)	Found shared object in library folder (2)
23-08-2017	Signal	v4.9.9	✓	×
28-02-2018		v4.16.9	✓	✓
06-08-2018		v4.24.8	✓	✓
09-02-2019		v4.33.5	✓	✓
09-08-2019		v4.45.2	✓	✓
12-02-2020		v4.55.8	✓	✓
20-08-2020		v4.69.4	✓	✓
18-02-2021		v5.4.6	✓	✓
20-08-2021		v5.21.5	✓	✓
18-02-2022		v5.32.7	✓	✓
05-08-2017	Telegram	v4.2.2	×	✓
19-02-2018		v4.8.4	×	✓
30-08-2018		v4.9.1	×	✓
09-02-2019		v5.3.1	×	✓
24-08-2019		v5.10.0	×	✓
16-02-2020		v5.15.0	×	✓
16-08-2020		v7.0.0	×	✓
18-02-2021		v7.4.2	×	✓
07-08-2021		v7.9.3	×	✓
14-02-2022		v8.5.2	×	✓
11-08-2017	WhatsApp	v2.17.296	✓	✓
09-02-2018		v2.18.46	✓	✓
18-08-2018		v2.18.248	✓	✓
08-02-2019		v2.19.34	✓	✓
07-08-2019		v2.19.216	✓	✓
13-02-2020		v2.20.22	✓	✓
05-08-2020		v2.20.196.16	✓	×
06-02-2021		v2.21.3.13	✓	×
09-08-2021		v2.21.17.1	✓	×
17-02-2022		v2.22.4.75	✓	×

Table 5.11: The frequency of Signal, Telegram, WhatsApp and SQLite library version releases, since 2017.

Codebase	Average Release time (in days) over the last five years
WhatsApp	6.324
Telegram	14.917
Signal	7.319
SQLite	39.48*

* Releases here reflect changes in the library source code. These updates do not necessarily imply changes in the API and are not mandatory for applications that use it.

5.5 Maintaining app stability

While collecting app-specific artefacts from memory is critical for accurate forensic timeline generation, apps enhanced with JIT-MF must remain usable by the device owner. Results from previous experiments (Sections 5.2.2 and 5.3.3) have shown that JIT-MF, while within a lab setup, performance overheads may not appear high. This may not be the case in a real-life setting where app usage may vary significantly. Furthermore, experimentation revealed possible UI degradations (Section 5.3.3) can be present when using specific JIT-MF Drivers. While this was not observed when using SQLite API-based JIT-MF Drivers during the experiment described in the previous section (Section 5.4), one cannot assume this is the general case. Different apps may leverage other APIs to execute application functionality of interest, whose resulting performance when equipped with the respective API-based JIT-MF Driver does not follow from SQLite API-based JIT-MF Drivers as used for the messaging apps used in the aforementioned case studies. Furthermore, app usage in a real-world rather than a lab setting may result in performance degradation, even in the case of SQLite API-based JIT-MF Drivers.

Even in such case, *Trigger_point* sampling can effectively maintain app stability by minimising overhead performance costs, without adversely affecting timeline accuracy. Possible performance overhead costs comprise storage costs impacting the device resources and app performance degradation due to a high frequency of memory dump triggering.

In the case of JIT-MF tools, storage costs can be attributed to two factors: i) the size of memory fragments dumped; and ii) the number of times a *Trigger_point* is invoked, causing the *Trigger_point* (*TP*) callback to be executed and a memory dump to be taken (*TP Frequency*), which can be limited to an extent by declaring a *Trigger_predicate*().

The experiment described in this section aims to determine how sampling methods can be used to aid *Trigger_predicates* in reducing the frequency with which *Evidence_object*s are dumped from memory to *JIT-MF_Logs*. Simultaneously an analysis is carried out to verify which sampling methods and parameters are optimal in reducing performance overheads while maintaining accuracy in the outputted app-specific artefacts comprising attack steps.

5.5.1 *Trigger_point* sampling methods

The size of memory fragments dumped is reduced significantly when opting for online collection within a JIT-MF Driver, rather than a full process memory dump. Yet, the burden on the device storage resources can still accumulate to large quantities depending on the TP frequency (*Trigger_point* Frequency).

Table 5.12: Sampling methods.

Sampling Method	**Implementation**	**Sampling Window**

Periodic Sampling	
	```
	bool sampling_predicate() {
	    current_time = get current time;
	    get current_second from current_time;

	    if (current_second % sample_window_value == 0):
	        reset random value;
	        return true;
	    else:
	        return false;
	}
	```
		A period of time (minutes)

Systematic Sampling	
	```
	bool sampling_predicate() {
	    tp_hit_counter = get current tp_hit_counter;

	    if (tp_hit_counter % sample_window_value == 0):
	        reset random value;
	        return true;
	    else::
	        return false;
	}
	```
		App dependent number of TP hits (number)

The two sampling methods shown in Table 5.12 are proposed to minimise the number of times a *Trigger_point* is invoked, thus reducing TP frequency and overall storage costs over time. A *sampling window* is defined as the range of possible values from which a sampling value is randomly chosen to determine whether the *Trigger_point* callback is executed. *Periodic sampling* refers to a generic sampling approach, using the current time to determine whether or not a memory dump should be taken. A random value in seconds (sampling value) is selected from an acceptable range of minutes (sampling window), and depending on whether or not the current time in seconds is exactly divisible by the randomly selected sampling value; the *Trigger_point* callback is executed. *Systematic sampling* is more specific to the JIT-MF framework, as it uses a counter storing the number of times the chosen *Trigger_point* was executed/hit at runtime (TP Hit) to determine whether or not a memory dump should be taken. A random counter value (sampling value) is selected, from an acceptable range of numbers representing TP hits (sampling window), and depending on whether or not the current counter value matches the randomly selected sampling value, the *Trigger_point* callback is executed.

When implementing either approach, selecting the sampling window size requires insight into the device owner's app usage pattern; however, *systematic sampling* requires more app code comprehension effort. Specifically, the JIT-MF Driver developer must gauge how often a *Trigger_point* is invoked in an acceptable time range (assuming typical app usage involving message sending and loading). For instance, if a chosen *Trigger_point* is hit very frequently during the app's runtime upon a single messaging event, then a small sampling window; e.g. a range of possible values between zero to five TP hits, would have little to no effect in reducing the burden JIT-MF Drivers have on running apps. In each method, once *Trigger_point* callback is executed, a new random value is selected, and in the case of *systematic sampling*, the counter is reset to zero.

While the primary motivation behind introducing sampling is eliminating app crashes and reducing the amount of storage required on the device to render JIT-MF forensically-enhanced apps as usable as their original counterparts, this is expected to come at a cost. When sampling is adopted, *Trigger_point* callback functions are only executed when the *sampling_predicate*() function in a JIT-MF Driver returns true, based on a random sampling value selected from a sampling window. This means that with an increasing sampling window, the number of *Trigger_point* callbacks executed decreases, resulting in fewer calls to dump app-specific *Evidence_object*s from memory. This allows for JIT-MF Drivers to fall prey to adversarial tactics. Since the random value generated for periodic sampling may not be a true random value, depending on the number of sources for randomness that the phone has, this creates a possibility whereby attackers carefully time messaging hijack attacks so that they occur outside the sampling value selected.

Furthermore, a large sampling window (e.g. a range between zero and thirty minutes) may lend itself useful to an attacker; since a large sampling window means that an attack step has larger periods during which no *Trigger_point* callbacks are executed. In the case of systematic sampling, if the sampling window is set to a high value because the typical usage behaviour of the user on the app calls for such a value, an adversarial actor may monitor the app usage to schedule the attack in a period when the app has been comparatively quiet. Therefore, even if TP hit count is relatively low, the attack step can still be missed.

Nevertheless, results from recent studies [4, 19] show that garbage collection algorithms available on Android devices, allow for the complete reconstruction of even complex objects from userland memory, in some cases much after the creation of said objects. Therefore, while objects from memory are collected less frequently when adopting sampling, this does not necessarily reflect in less *Evidence_object*s collected. Garbage collection algorithms running on the device should work in JIT-MF's favour, such that *Evidence_object*s not immediately retrieved due to sampling may still be in memory when the *sampling_predicate*() function returns true and the *Trigger_point* callback function is eventually executed. Hence, while a decrease in the number of messaging objects collected is expected, preliminary results indicate that this decrease will be minimal.

5.5.2 Experiment setup

JIT-MF Drivers with *Evidence_object*s selected from the app layer and *Trigger_point*s from the native layer were shown to produce accurate *JIT-MF_Logs* (Section 5.2.2) and forensic timelines (Section 5.3.3). Yet these drivers were observed to have the worst performance overheads when used with Pushbullet and Telegram apps (Section 5.3.3). The same type of driver was also used for Signal (Section 5.3.3), with similar observations regarding UI degradation. Therefore, this experiment was carried out using these JIT-MF Drivers (with app-layer *Evidence_object*s and native-layer *Trigger_point*s shown in Listing A.1 - Listing A.6 found in Appendix A) and the Pushbullet (v18.4.0), Telegram (v5.12.0), and Signal (v5.18.5) apps. The apps enhanced with JIT-MF Drivers were installed on two Google Pixel 3XL emulators running Android 10 (API 29). An emulator was used for ease of automating message sending. While specific drivers and apps are used to conduct this experiment, the sampling methods are stack layer and app-agnostic. Therefore, the insight gained from the results of this experimentation can be transferred to other scenarios with different apps and JIT-MF Drivers with *Evidence_object*s and *Trigger_point*s selected from other layers, for which performance degradation is observed.

This experiment aims to observe how random sampling values selected from increasing sampling windows, for each sampling method proposed in Table 5.12, affect the app's performance at runtime (in terms of stabilisation; i.e. the number of crashes and device storage resources) under typical app usage. For the scope of this experiment, typical messaging functionality comprises chat loading and message sending carried out via `adb input` keystrokes in the case of Telegram and Signal and through Selenium for Pushbullet. A sampling window range is set per sampling method. This range defines the maximum and minimum values of the sampling windows considered per sampling method. In the case of periodic sampling, the sampling window ranges between zero to five minutes. This value is based on the time taken for the experiment outlined above to be carried out. For systematic sampling, preliminary observations were carried out to understand how many times the specific *Trigger_point* chosen for an app is hit in five minutes. The sampling window ranges were between one TP hit to 5000, 400 and 32000 TP hits for Pushbullet, Telegram, and Signal, respectively.

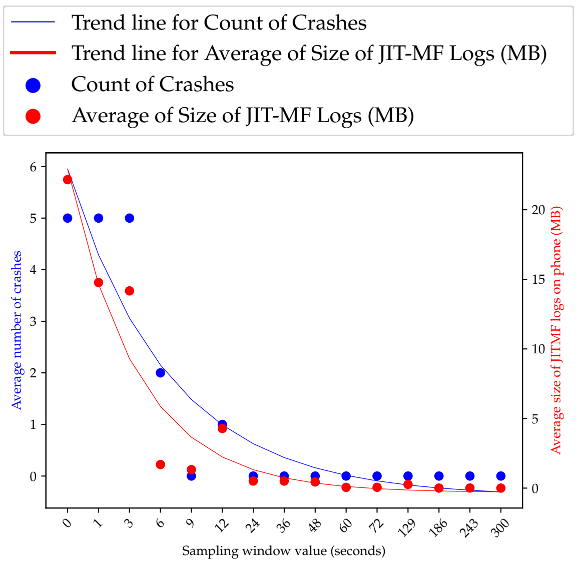
Performance degradation metrics. The `logcat` crash buffer determines whether or not the app crashes during either run. Statistics related to Janky frames were also gathered to determine any slow interactions with the app's GUI. Janky frame statistics can only be generated while the app is running; therefore, if Janky frame statistics are not outputted, this is an indicator that the app has crashed. The accumulated size of *JIT-MF_Logs* is used to measure the impact on the device's storage capacity.

The experiment was carried out as follows. Preliminary runs were used for each app to determine the interval between the different sampling windows used, up to the maximum value in the sampling window range, per sampling method. A JIT-MF Driver was created for each sampling window with the relevant implementation of the *sampling_predicate*() function. The *sampling_predicate*() function can be implemented within the *Trigger_predicate*() function in a JIT-MF Driver or as a separate function and called from the *Trigger_predicate*() function. Any residue logs are deleted, and Janky frame statistics are reset. The respective driver is pushed on the device. A chat is loaded and a pseudo-random message with the same prefix is sent at random five times, every ten seconds. This process is repeated five times for each to reach convergence. Preliminary runs indicated that when an app hangs/crashes during the experiment run described above, it never "recovers" unless it is closed and reopened. Therefore, the number of messages sent was kept at par with manual app usage.

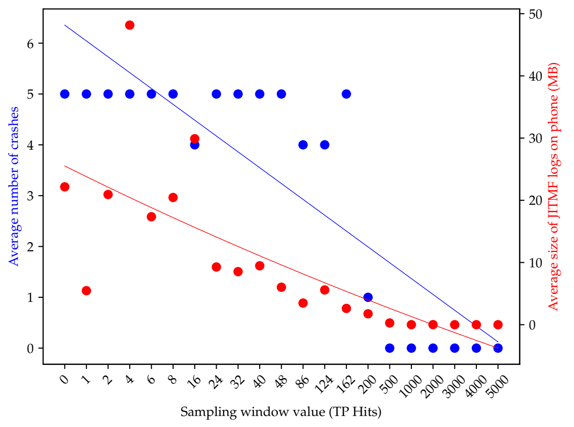
5.5.3 Results

Figure 5.3(a), Figure 5.3(c), Figure 5.3(e) present the results obtained when using *periodic sampling*, while figures Figure 5.3(b), Figure 5.3(d), Figure 5.3(f) present those obtained when using *systematic sampling*, for Pushbullet, Telegram and Signal, respectively. In each of the figures, the number of crashes (blue) and total storage taken up on the device by JIT-MF dumps (red), are plotted against varying sampling values (up until the maximum sampling window value set in the experiment).

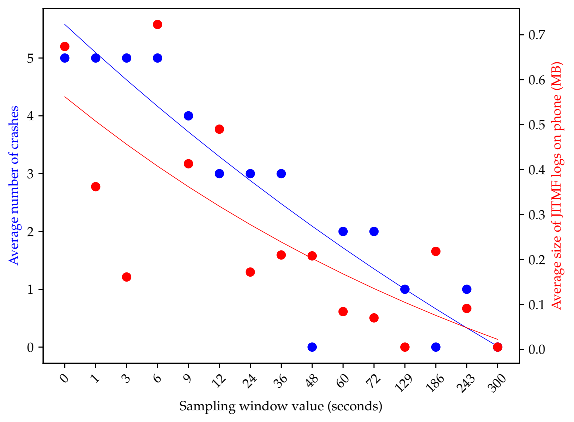
As expected, across both sampling methods, the increase in sampling window causes the number of crashes and storage required to reduce significantly until a plateau is reached. Furthermore, while the number of crashes becomes negligible, the size of the output JIT-MF dumps is higher than 0MB, meaning that even with an increase in the sampling window, the output is still produced, and, JIT-MF is still effective in dumping *Evidence_object*s in memory. Table 5.13 shows the percentage of events retrieved, for the sampling window where it is evident from the graphs presented in Figure 5.3, that a plateau has been reached. Results from this table show that, even when a large sampling window is adopted, most ground truth events are still found in *JIT-MF_Logs*, demonstrating that the Garbage Collector is working in JIT-MF's favour. This bodes well for the applicability of JIT-MF in a realistic scenario, especially when considering the improvement of the app's performance in terms of the number of crashes.



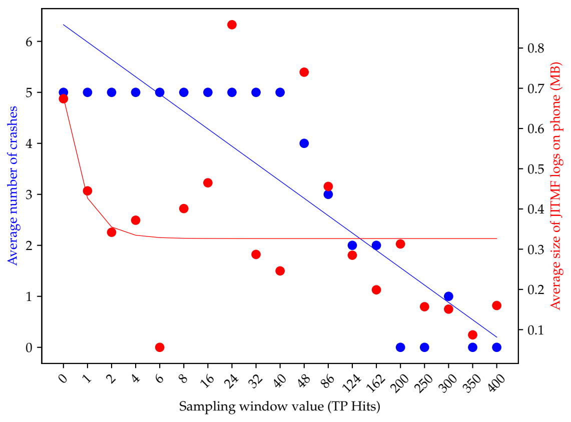
(a) Pushbullet: Average number of app crashes and storage requirements (MB) against increasing periodic sampling window (in seconds).



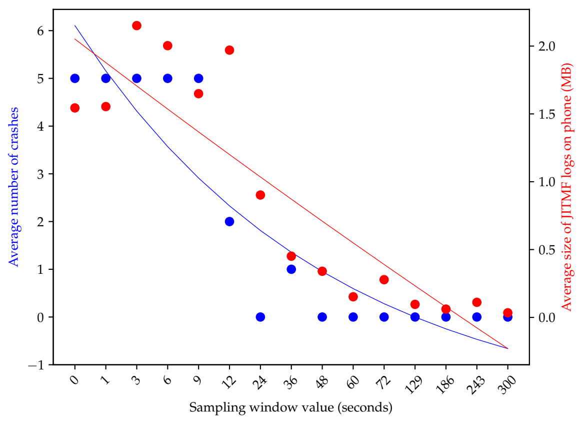
(b) Pushbullet: Average number of app crashes and storage requirements (MB) against increasing systematic sampling window (in number of TP hits).



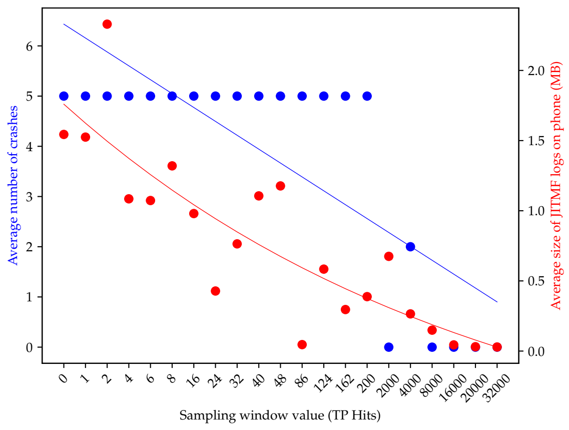
(c) Telegram: Average number of app crashes and storage requirements (MB) against increasing periodic sampling window (in seconds).



(d) Telegram: Average number of app crashes and storage requirements (MB) against increasing systematic sampling window (in number of TP hits).



(e) Signal: Average number of app crashes and storage requirements (MB) against increasing periodic sampling window (in seconds).



(f) Signal: Average number of app crashes and storage requirements (MB) against increasing systematic sampling window (in number of TP hits).

Figure 5.3: Sampling results obtained for Signal, Telegram and Pushbullet

Table 5.13: % Ground truth timeline events captured per performance-optimal sampling window size.

App	Sampling Method	Optimal Sampling Window	% Ground Truth Collected Events
Pushbullet	Periodic	9s	86
	Systematic	500 TP hits	58
Telegram	Periodic	48s	62
	Systematic	200 TP hits	70
Signal	Periodic	24s	92
	Systematic	8000 TP hits	92

The amount of storage on the SD Card is 512MB. Therefore, at worst, *JIT-MF_Logs* took ∼4% of the storage available. While there is a correlation between crashes and the total percentage of storage taken (0.28 and 0.44 when using systematic and periodic sampling, respectively), this is not strong. Therefore the crashes are most likely resulting from IO operations rather than the increase in the usage of storage resources.

Results from Figure 5.3 show that both sampling methods were equally successful in significantly reducing overhead costs. That is, a specific sampling window size is identifiable in each of the graphs, for which the number of crashes and storage required on the device is the same for both sampling methods per app. However, a distinction between the two methods can be made regarding the implementation effort required for each method and the likelihood of each sampling method falling prey to adversarial tactics.

Sampling implementation effort. Periodic sampling requires a few more computational steps, as it involves obtaining the current time, translating it to seconds and then checking if it is exactly divisible by the random value of seconds selected, as opposed to systematic sampling, where only the *Trigger_point* hit counter is needed and checked against a random value. Nevertheless, while periodic sampling can be implemented without prior knowledge of the app's usage pattern or understanding of how the app functions, the same cannot be said for systematic sampling. The latter sampling method is based on the number of times a *Trigger_point* is hit. While the measure of time is constant across apps and different scenarios, the number of times a *Trigger_point* is hit varies depending on how the user uses an app and how it functions. Therefore, a preliminary exercise must be done before applying this sampling method, whereby the average number of *Trigger_point* hits in a time range is determined, per app, scenario, and usage pattern.

Risk of adversarial tactics. While both sampling methods rely on randomness to generate a value within a sampling window, the sources of randomness for a specific device differ on each device and are independent of implementation. In the case that an attacker can determine the random value, an attacker can perform a messaging hijack attack during the portion of the sampling window where the *sampling_predicate*() criterion is not met and a memory dump is not taken; i.e. at the wrong time or not the right amount of TP hit count. From the attacker's point of view, the periodic sampling method would be easiest to evade since it requires no additional knowledge except for a measure of time itself. The systematic sampling method would require more effort for the same reason that it requires more implementation effort. It may be more difficult for the attacker to determine the current value of the *Trigger_point* hit counter, especially without knowing how the app is being used. While an attacker may still circumvent this sampling method by monitoring periods of inactivity, it raises the bar for the attacker requiring increased sophistication to carry out the attack.

5.6 Summary of Experimental Findings

This chapter leveraged the JIT-MF framework proposed in Chapter 4 to determine the least level of invasiveness required for timely and accurate acquisition of app-specific artefacts from memory related to app hijack attack steps. Using different implementation approaches to the JIT-MF framework, this exploration comprised finding the optimal positioning of JIT-MF *Trigger_point*s and *Evidence_object*s within the Android technology stack. The objective of the experiments discussed in this chapter was to determine the accuracy of the app-specific artefacts collected while minimising the level of app invasiveness required to obtain these artefacts from memory, contributing to increased app comprehension efforts and app stability as much as possible. This was achieved by progressively selecting *Trigger_point*s and *Evidence_object*s from lower levels of the stack. The experimental methodology used a realistic simulation of messaging app (IM and SMS) hijack attacks and leveraging JIT-MF to obtain *JIT-MF_Logs* as an additional source of evidence. JIT-MF Drivers and Runtime were installed at the app level: i) to avoid device-invasive methods (e.g. rooting), and ii) since the app level is the most obvious choice for collecting app-specific artefacts required for remediating app hijack attacks. *Trigger_point*s and *Evidence_object*s were selected from different layers of the Android technology stack to identify how the *JIT-MF_Logs*' accuracy is affected when selecting these critical JIT-MF components from lower layers in the stack that are less app invasive. The execution of simulated attacks and background traffic, along with the collection and analysis of *JIT-MF_Logs*, is completely automated within a setup comprising emulators and actual Android devices.

Table 5.14 summarises the experiments carried out during this exploration within the setting of messaging hijack attacks. The table outlines each experiment (denoted by the section in which it was described) and its main objectives concerning the hypothesis presented in this thesis (that is, that timely and non-invasive logging of hijacked functionality is possible through process memory introspection). The stack layers explored within each experiment, for both *Evidence_object*s and *Trigger_point*s selection, are listed along with the main conclusions and impact on follow-up experiments.

Overall, these results show that the timely and minimally invasive logging of hijacked functionality is possible through i) trigger-based memory dumps containing app-specific artefacts as implemented by JIT-MF *Trigger_point*s and *Evidence_object*s in JIT-MF Drivers and ii) the implementation of JIT-MF at app level using *Trigger_point*s and *Evidence_object*s from the API layer of the Android technology stack. Specifically, the API layer of the Android technology stack was shown to be the optimal layer for *Trigger_point*s and *Evidence_object*s selection as: i) resulting *JIT-MF_Logs* maintained accuracy; ii) minimal invasiveness was required for selection due to publicly available documentation of APIs; and iii) these JIT-MF Drivers resulted in observable stable app performance. An initial comparison of the generated forensic timelines using *JIT-MF_Logs* was made, with baseline forensic sources to establish the accuracy of lower level *Trigger_point*s in generating forensic timelines for messaging hijack investigations. However, results showed that the precision of the generated timelines is low, possibly due to further post-processing and analysis required to identify *Evidence_object*s directly related to attack steps. This hinders the ability of *JIT-MF_Logs* to aid in complete and accurate attack steps recovery (from malware entry point), as required for incident response which has yet to be demonstrated.

Table 5.14: Summary of experiments carried out for JIT-MF positioning exploration.

Experiment	**Objective**	**Stack layer**		**Main conclusions**	**Impact**
		Evidence_object	*Trigger_point*		
Section 5.2	Comparison of accuracy in the attack steps recorded and the associated storage overhead costs when selecting *Trigger_point*s from different layers in the Android technology stack.	App	App, API, Native, Native with device-level predicates	• *Trigger_point*s selected from lower levels in the technology stack can be as effective and efficient as app-specific ones. • Offline and online collection methods have very similar results. Yet offline collection requires significantly more storage space on the device on average.	• Selecting JIT-MF *Trigger_point*s at lower levels is sufficient for collecting app-specific artefacts. • Opting for online collection. • Requires further experimentation concerning app stability.
Section 5.3	Assessment of the added accuracy of forensic timelines generated using *JIT-MF_Logs*.	App	Native, API	• JIT-MF Drivers can produce accurate forensic timeline sequences with sufficient detail one would normally expect only from developer-provided app logs.	• Selecting *Trigger_point*s at lower levels produces accurate timelines, but JIT-MF Driver development is still app invasive due to app-specific *Evidence_object*. • Requires further experimentation concerning app stability.
Section 5.4	Comparison of accuracy in the attack steps recorded when selecting *Trigger_point*s and *Evidence_object*s from lower layers in the technology stack.	API	API	• API-based JIT-MF Drivers require less frequent updates due to fewer codebase updates. • API-level *Trigger_point*s and *Evidence_object*s require minimal invasiveness, as they rely on publicly-available API documentation. • JIT-MF Driver can be generalised without compromising timeline accuracy by basing *Trigger_point*s and *Evidence_object*s on widely adopted APIs.	• Multiple apps can use the same API-based JIT-MF Driver. • App-specific usage of API requires some parsing of collected *JIT-MF_Logs*.
Section 5.5	Maintaining app stability while timely collecting app artefacts from memory.	App	Native	• JIT-MF Drivers with Native-level *Trigger_point*s require sampling to avoid app performance degradation comprising app crashes. • Optimal *Trigger_point* sampling results in reduced *JIT-MF_Logs* size, which may still contain app-specific artefacts.	• Two possible sampling strategies are made available. JIT-MF Driver developers can perform a parameter analysis to see which parameters fit their app-attack scenario best.

6 Evaluation of JIT-MF with the State-of-the-Art

In the previous chapters, JIT-MF was proposed as a framework that can timely collect app-specific evidence from memory relating to possibly hijacked benign apps. Furthermore, several JIT-MF implementation approaches were used to explore the hypothesis of whether timely and accurate dumping of *Evidence_object*s related to attack steps, can be carried out while incurring minimal invasiveness. Thus, the previous chapter detailed an exploration of JIT-MF positioning within the Android technology stack, which demonstrated that the selection of *Evidence_object*s and *Trigger_point*s from the API layer contributes towards timely and non-invasive logging of app-specific artefacts from memory that represent hijacked attack steps. Yet, to address the main hypothesis, further experimentation is required to evaluate the ability of *JIT-MF_Logs* to enable the complete reconstruction of attack steps during a forensic investigation. This chapter explores this in the context of messaging hijack attacks and state-of-the-art forensic tools, thus addressing the third objective ***O3*** of this thesis. Experimentation is conducted by leveraging SQLite API-based JIT-MF Drivers that were shown to be non-invasive with respect to the stack layer comprehension required for both *Trigger_point* and *Evidence_object* selection, without compromising forensic timeline accuracy.

Several forensic tools are used at different stages of the incident response cycle, each fulfilling different purposes. At the preparation stage, Endpoint Detection and Response (EDR) tools can help monitor and record events occurring in real time on devices. After an incident occurs, Mobile Forensic tools are used to extract and analyse data from mobile devices. Their focus is on normalising data from different forensic sources stored on the device to present the investigator with a single chronological timeline of events.

Just-in-Time Memory Forensics can contribute to real-time evidence collection from memory and produces *JIT-MF_Logs* as an additional forensic source. The purpose and value of *JIT-MF_Logs* can only be evaluated along with other sources collected and analysed by EDR and mobile forensic tools. To this end, this chapter aims to demonstrate how *JIT-MF_Logs* improve state-of-the-art mobile forensics in terms of collected hijacked app steps in the case of real-world app hijack case study involving messaging apps (Section 6.1). Furthermore, a complete anomaly detection setup shows that the evidence found in *JIT-MF_Logs* is detectable as anomalous and carries sufficient features to be correlated with the attack's entry point on the device (Section 6.2). Therefore, using a detection and correlation algorithm that leverages evidence uniquely found in *JIT-MF_Logs*, the complete and precise forensic timeline of app hijack attack steps can be generated, and malware entry points can be identified. * "

6.1 Evaluating JIT-MF in combination with mobile forensic tools

This experiment aims to demonstrate how evidence of a realistic app hijack attack found in *JIT-MF_Logs* can uniquely contribute to forensic timelines generated by state-of-the-practice mobile forensic tools by collecting evidence of hijacked messaging attack steps in the form of message objects that are not found in any other forensic source or collected by any other forensic tool.

6.1.1 MobFor: A JIT-MF tool

MobFor is an open-source and publicly available[1] JIT-MF tool that was created for this experiment. The tool has multiple JIT-MF Drivers, including those targeting the WhatsApp messaging app. MobFor uses an app-invasive approach to install JIT-MF Drivers and Driver Runtime in targeted apps, which involves app repackaging. For this experiment, the JIT-MF Driver used comprised SQLite API-based driver.

6.1.2 Incident scenario

The incident scenario follows from the motivational case study presented in Section 3.3. A target victim's WhatsApp app on their Android device was forensically enhanced through MobFor. They have been at the receiving end of a social engineering phishing campaign and have unknowingly installed the WhatsApp Pink malware on their Android device. The malware propagates by automatically replying to incoming legitimate WhatsApp messages with a download link to the malware itself akin to a message-proxying attack.

[1]`https://gitlab.com/mobfor/mobfor-project`

Table 6.1: Forensic tools used.

Forensic Tool	Version	Description
MSAB XRY	XRY Office v9.6	Proprietary (Trial Version)
Belkasoft	1.10.8387	Proprietary (Trial Version)

Evolved malware for stealth. The original WhatsApp Pink malware conceals its actions by leveraging messaging apps to propagate and even hides the app's icon from the home screen. Yet the propagated message by the malware is still visible in the chat.

The WhatsApp Pink malware was enhanced further to increase its stealth by enabling it to remove any forensic footprints of the attack steps. Any messages propagated by the stealthier version of WhatsApp Pink are deleted from the owner's phone, leaving no visible trace of the malware's actions, as shown Figure 3.2. `AndroidViewClient`[2] was used to simulate stealthy attack steps comprising the deletion of messages and uninstallation of the WhatsApp Pink malware. This enables the malware to operate with maximum stealth by leaving no trace of the malicious app, further reducing its forensic footprint.

6.1.3 Mobile forensic tools setup

Responding to an incident relies on three main steps: i) Collection, ii) Parsing, and iii) Analysis of evidence to produce a timeline of events. Existing forensic tools are typically equipped with collection and parsing features, enabling an incident responder to analyse the forensic timeline produced through the available tools. Two mobile forensic tools are used in this experiment, to demonstrate how MobFor can contribute to existing digital forensics tools in the incident scenario described: Belkasoft Evidence Centre X and MSAB's XRY (see Table 6.1). Table 6.2 describes how each tool was configured regarding the sources gathered during the collection phase, the parsing tool used, and the tool used to generate a forensic timeline for analysis.

Mobile forensic tools evidence collection. Although not publicly disclosed, one way with which mobile forensic tools (including Belkasoft and XRY) collect private app data (found in internal storage containing decrypted database files on disk) is through the use of `adb backup`. Since newer app versions typically have their custom `BackupAgent` (defined in the app's Manifest file), this process first requires an app downgrade which calls for app uninstallation. Open-source forensic collection tools like WhatsApp DB

[2]`https://github.com/dtmilano/AndroidViewClient`

Table 6.2: Digital investigation configurations.

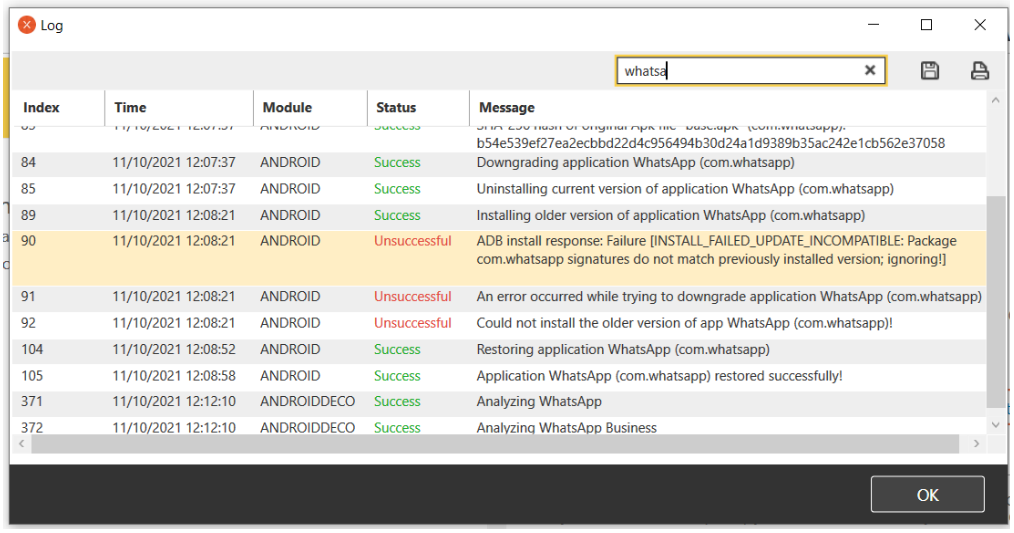
Incident Response step	**XRY**	**Belkasoft**	**MobFor**
Collection	• XRY Agent • Additional forensic sources	• Belkasoft Agent • Additional forensic sources	• *JIT-MF_Logs*
Parsing	XRY XAMN	Belkasoft	*JIT-MF_Logs* MobFor Parser
Analysis	Timesketch		

Extractor[3] [128, 129] enable the uninstallation of apps, without deleting private app data containing texts using the following command: `adb uninstall -k com.whatsapp`. While this solution has been shown to work, it is less stable on newer versions of Android and may require the complete uninstallation of the app in some cases, causing the loss of data and potential evidence.

This evidence extraction method, however, cannot be used simultaneously with MobFor since the forensically enhanced version of the targeted benign app carries a different signature than the original one, as shown in Figure 6.1. The partial uninstallation of the enhanced app and the older app's signature mismatch cause Android to produce an error, preventing the older app from being installed. Therefore, when implementing JIT-MF using app repackaging, changes to the app must include modifications to the Manifest file to set `debuggable=true`. This allows private files to be stored on external storage so that mobile forensic tools may eventually collect them. The result is equivalent to the `adb backup` method. When implementing JIT-MF using app-level virtualisation, the backup capabilities of the app rely on the container app. Regardless, the same approach of modifying the Manifest file can be used to obtain private files related to plugin app data.

Mobile forensic tool setup. While the version of tools used were trials, this did not impact the experimentation results. Most of the functionality withheld from trial versions is related to the availability of rooting exploits and iOS features. Since the experimentation scenario involves Android OS and the device belongs to a victim who intends to continue using it, rooting is considered a non-viable option. Therefore these features were not needed. The Belkasoft trial version presented limitations regarding the amount of data from findings that could be exported to a file format. However, all the findings are still available through the user interface.

[3] `https://github.com/EliteAndroidApps/WhatsApp-Key-DB-Extractor`

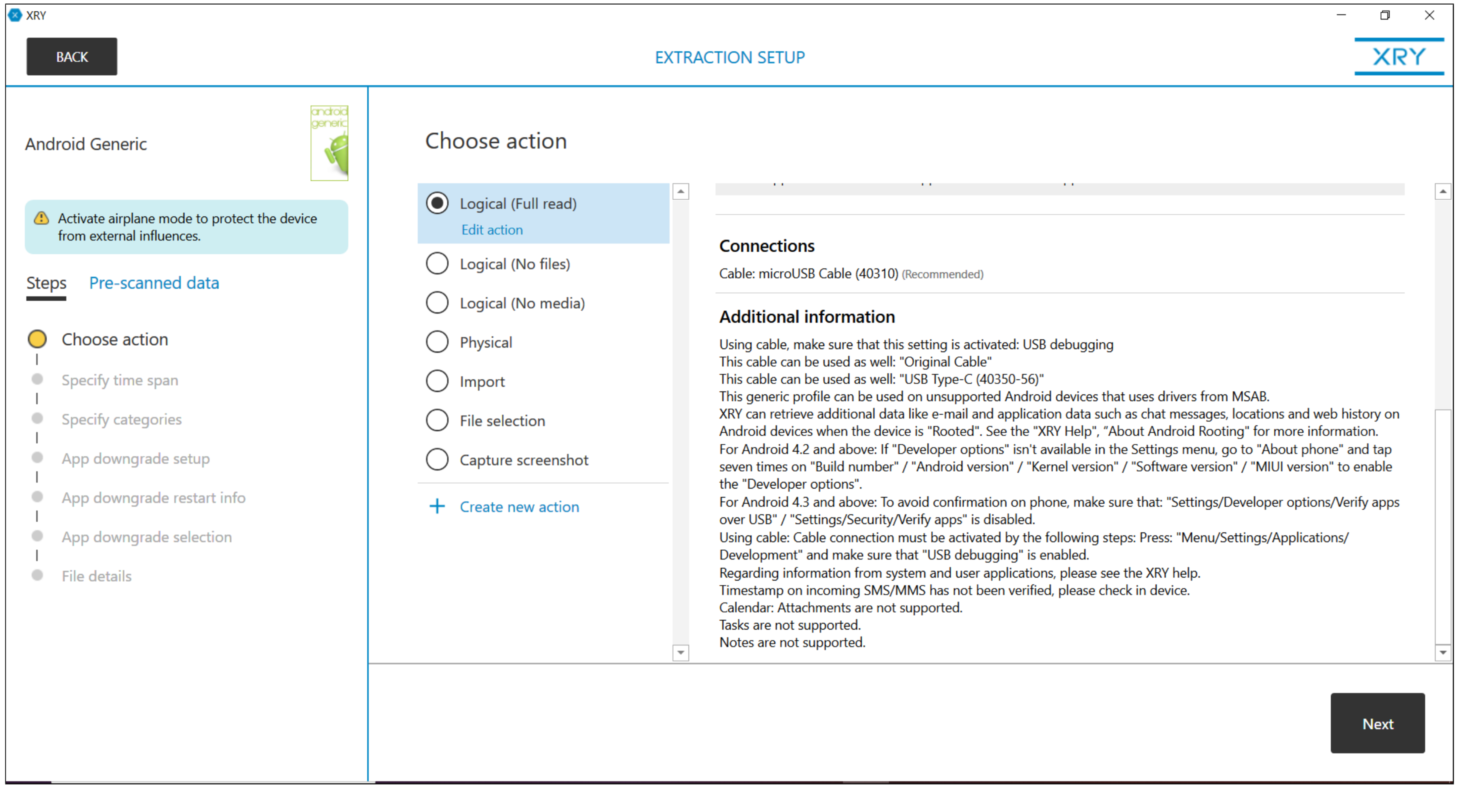


Index	Time	Module	Status	Message
[illegible]	11/10/2021 12:07:37	ANDROID	Success	SHA-256 hash of original Apk file 'base.apk' (com.whatsapp): b54e539ef27ea2ecbbd22d4c956494b30d24a1d9389b35ac242e1cb562e37058
84	11/10/2021 12:07:37	ANDROID	Success	Downgrading application WhatsApp (com.whatsapp)
85	11/10/2021 12:07:37	ANDROID	Success	Uninstalling current version of application WhatsApp (com.whatsapp)
89	11/10/2021 12:08:21	ANDROID	Success	Installing older version of application WhatsApp (com.whatsapp)
90	11/10/2021 12:08:21	ANDROID	Unsuccessful	ADB install response: Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE: Package com.whatsapp signatures do not match previously installed version; ignoring!]
91	11/10/2021 12:08:21	ANDROID	Unsuccessful	An error occurred while trying to downgrade application WhatsApp (com.whatsapp)
92	11/10/2021 12:08:21	ANDROID	Unsuccessful	Could not install the older version of app WhatsApp (com.whatsapp)!
104	11/10/2021 12:08:52	ANDROID	Success	Restoring application WhatsApp (com.whatsapp)
105	11/10/2021 12:08:58	ANDROID	Success	Application WhatsApp (com.whatsapp) restored successfully!
371	11/10/2021 12:12:10	ANDROIDDECO	Success	Analyzing WhatsApp
372	11/10/2021 12:12:10	ANDROIDDECO	Success	Analyzing WhatsApp Business

Figure 6.1: XRY Log output showing WhatsApp downgrade failure.

Both XRY and Belkasoft were configured similarly. An agent-based collection was used by each respective tool, targeting logical collection (see Figure 6.2 and Figure 6.3). This comprised each tool installing proprietary apps (agent) on the device via USB connection, as part of the forensic collection process. For this experimentation, "Logical" collection with "Full read" capabilities was initiated for all tools (see Figure 6.2); that is, all data accessible to the forensic tool agent is collected, bar any data that requires device rooting. Whenever a collection step required rooting, it was skipped. In this case, data available to the agent comprised of Device data, Memory card data, and SIM card data, as shown in Figure 6.3.

Furthermore, each tool also accepts additional data sources that the analyst has in their possession to support the findings gathered by the tool. Once the sources are gathered, each tool parses them and presents them in a way that can be useful to the investigator, as shown in Figure 6.4. Figure 6.5 shows the parsed forensic artefacts from different sources (four in this case) gathered for forensic analysis. The first entry in the image comprises the findings collected by the tool's agent. The next one consists of private data collected from the app's internal storage (`/data/data` folder), obtained through its `debuggable=true` property set in the Manifest. A third entry consists of data gathered from an `adb backup` output, and finally, the last additional source consists of additional logs of use, in this case, `logcat` and `dumpsys`.



XRY

BACK

EXTRACTION SETUP

XRY

Android Generic

Activate airplane mode to protect the device from external influences.

Steps | Pre-scanned data

- Choose action
- Specify time span
- Specify categories
- App downgrade setup
- App downgrade restart info
- App downgrade selection
- File details

Choose action

- [✓ Logical (Full read)] Edit action
- Logical (No files)
- Logical (No media)
- Physical
- Import
- File selection
- Capture screenshot
- Create new action

Connections

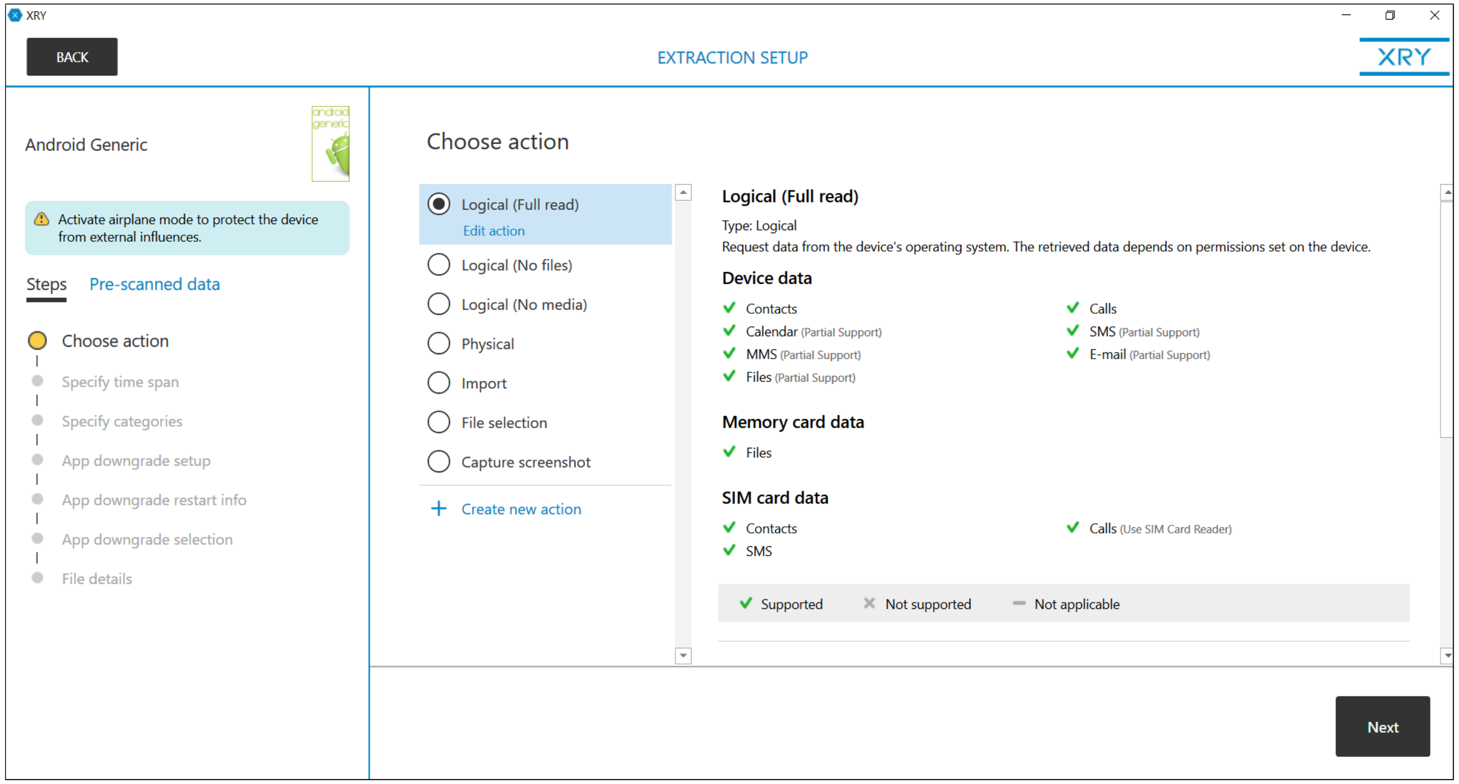
Cable: microUSB Cable (40310) (Recommended)

Additional information

Using cable, make sure that this setting is activated: USB debugging
This cable can be used as well: "Original Cable"
This cable can be used as well: "USB Type-C (40350-56)"
This generic profile can be used on unsupported Android devices that uses drivers from MSAB.
XRY can retrieve additional data like e-mail and application data such as chat messages, locations and web history on Android devices when the device is "Rooted". See the "XRY Help", "About Android Rooting" for more information.
For Android 4.2 and above: If "Developer options" isn't available in the Settings menu, go to "About phone" and tap seven times on "Build number" / "Android version" / "Kernel version" / "Software version" / "MIUI version" to enable the "Developer options".
For Android 4.3 and above: To avoid confirmation on phone, make sure that: "Settings/Developer options/Verify apps over USB" / "Settings/Security/Verify apps" is disabled.
Using cable: Cable connection must be activated by the following steps: Press: "Menu/Settings/Applications/Development" and make sure that "USB debugging" is enabled.
Regarding information from system and user applications, please see the XRY help.
Timestamp on incoming SMS/MMS has not been verified, please check in device.
Calendar: Attachments are not supported.
Tasks are not supported.
Notes are not supported.

Next

Figure 6.2: XRY Collection Method - 1 of 2. The 'Logical (Full read)' option is chosen, which collects all possible data from the device's file system under investigation.



Android Generic

Activate airplane mode to protect the device from external influences.

Steps | Pre-scanned data

- Choose action
- Specify time span
- Specify categories
- App downgrade setup
- App downgrade restart info
- App downgrade selection
- File details

Choose action

- [✓ Logical (Full read)] Edit action
- Logical (No files)
- Logical (No media)
- Physical
- Import
- File selection
- Capture screenshot
- Create new action

Logical (Full read)

Type: Logical

Request data from the device's operating system. The retrieved data depends on permissions set on the device.

Device data

✓ Contacts	✓ Calls
✓ Calendar (Partial Support)	✓ SMS (Partial Support)
✓ MMS (Partial Support)	✓ E-mail (Partial Support)
✓ Files (Partial Support)	

Memory card data

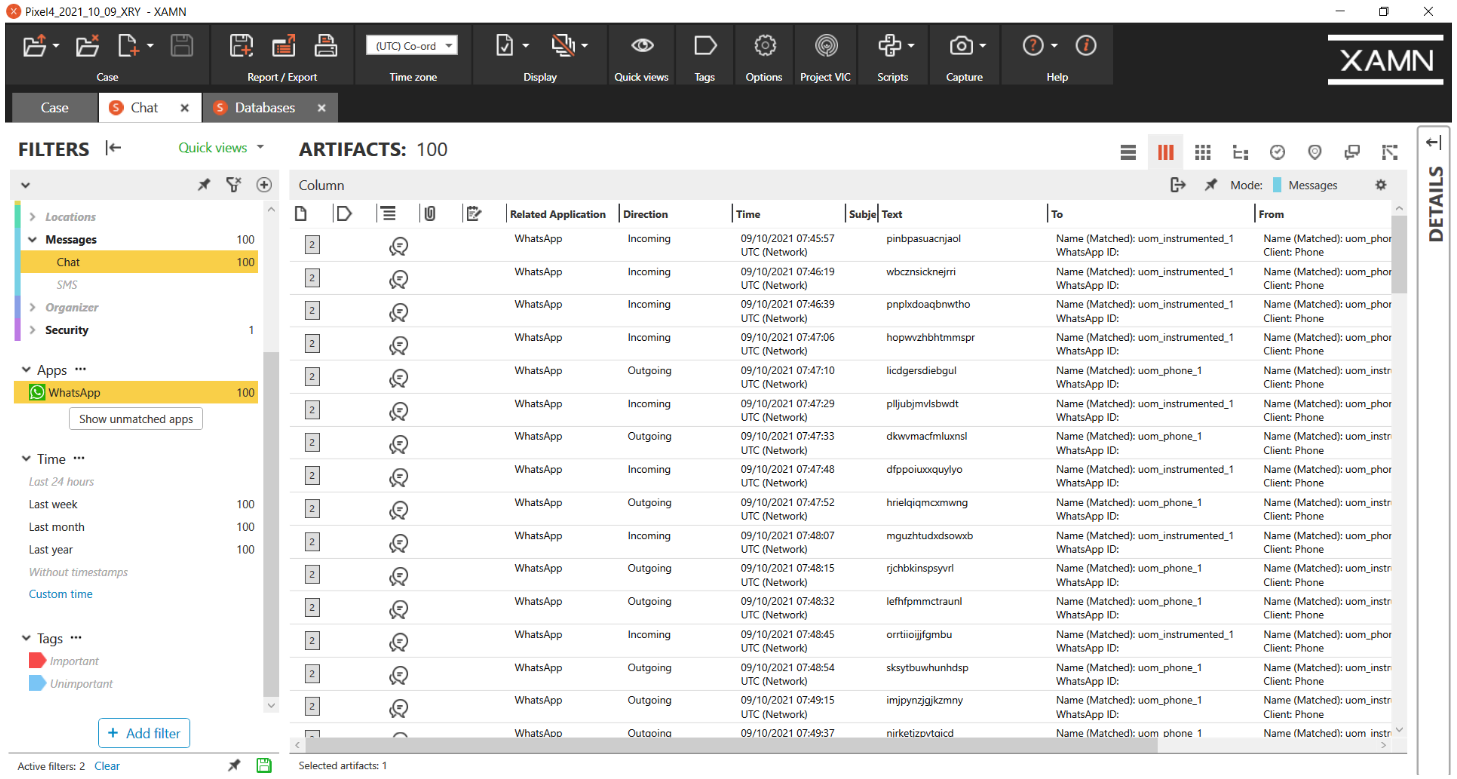
✓ Files

SIM card data

✓ Contacts	✓ Calls (Use SIM Card Reader)
✓ SMS	

✓ Supported ✕ Not supported — Not applicable

Figure 6.3: XRY Collection Method - 2 of 2. The image shows the data collected during the 'Logical (Full read)' acquisition.



Locations
Messages 100
Chat 100
SMS
Organizer
Security 1

Apps
WhatsApp 100
Show unmatched apps

Time
Last 24 hours
Last week 100
Last month 100
Last year 100
Without timestamps
Custom time

Tags
Important
Unimportant

Add filter

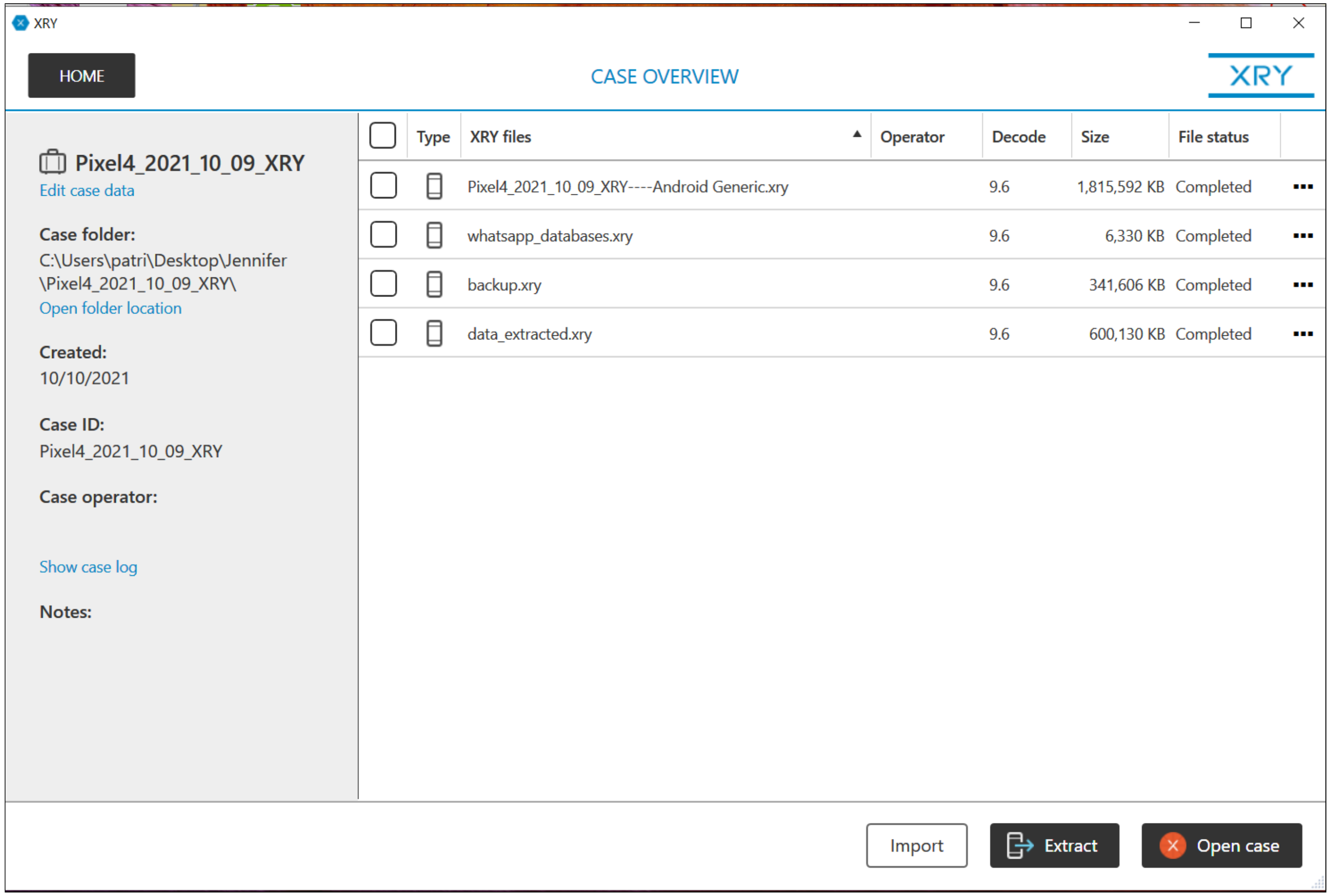
Active filters: 2 Clear

ARTIFACTS: 100

Related Application	Direction	Time	Subje	Text	To	From
WhatsApp	Incoming	09/10/2021 07:45:57 UTC (Network)		pinbpasuacnjaol	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Incoming	09/10/2021 07:46:19 UTC (Network)		wbcznsicknejrri	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Incoming	09/10/2021 07:46:39 UTC (Network)		pnplxdoaqbnwtho	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Incoming	09/10/2021 07:47:06 UTC (Network)		hopwvzhbhtmmspr	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Outgoing	09/10/2021 07:47:10 UTC (Network)		licdgersdiebgul	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Incoming	09/10/2021 07:47:29 UTC (Network)		plljubjmvlsbwdt	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Outgoing	09/10/2021 07:47:33 UTC (Network)		dkwvmacfmluxnsl	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Incoming	09/10/2021 07:47:48 UTC (Network)		dfppoiuxxquylyo	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Outgoing	09/10/2021 07:47:52 UTC (Network)		hrielqiqmcxmwng	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Incoming	09/10/2021 07:48:07 UTC (Network)		mguzhtudxdsowxb	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Outgoing	09/10/2021 07:48:15 UTC (Network)		rjchbkinspsyvrl	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Outgoing	09/10/2021 07:48:32 UTC (Network)		lefhfpmmctraunl	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Incoming	09/10/2021 07:48:45 UTC (Network)		orrtiioijjfgmbu	Name (Matched): uom_instrumented_1 WhatsApp ID:	Name (Matched): uom_phor Client: Phone
WhatsApp	Outgoing	09/10/2021 07:48:54 UTC (Network)		sksytbuwhunhdsp	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Outgoing	09/10/2021 07:49:15 UTC (Network)		imjpynzjgjkzmny	Name (Matched): uom_phone_1 WhatsApp ID:	Name (Matched): uom_instr Client: Phone
WhatsApp	Outgoing	09/10/2021 07:49:37		njrketizpvtgicd	Name (Matched): uom_phone_1	Name (Matched): uom_instr

Selected artifacts: 1

Figure 6.4: XRY Output containing sample WhatsApp messaging events.



Pixel4_2021_10_09_XRY
Edit case data

Case folder:
C:\Users\patri\Desktop\Jennifer\Pixel4_2021_10_09_XRY\
Open folder location

Created:
10/10/2021

Case ID:
Pixel4_2021_10_09_XRY

Case operator:

Show case log

Notes:

Type	XRY files	Operator	Decode	Size	File status
	Pixel4_2021_10_09_XRY----Android Generic.xry		9.6	1,815,592 KB	Completed
	whatsapp_databases.xry		9.6	6,330 KB	Completed
	backup.xry		9.6	341,606 KB	Completed
	data_extracted.xry		9.6	600,130 KB	Completed

Figure 6.5: XRY Additional Forensic Sources.

Table 6.3: Mobile device specifications.

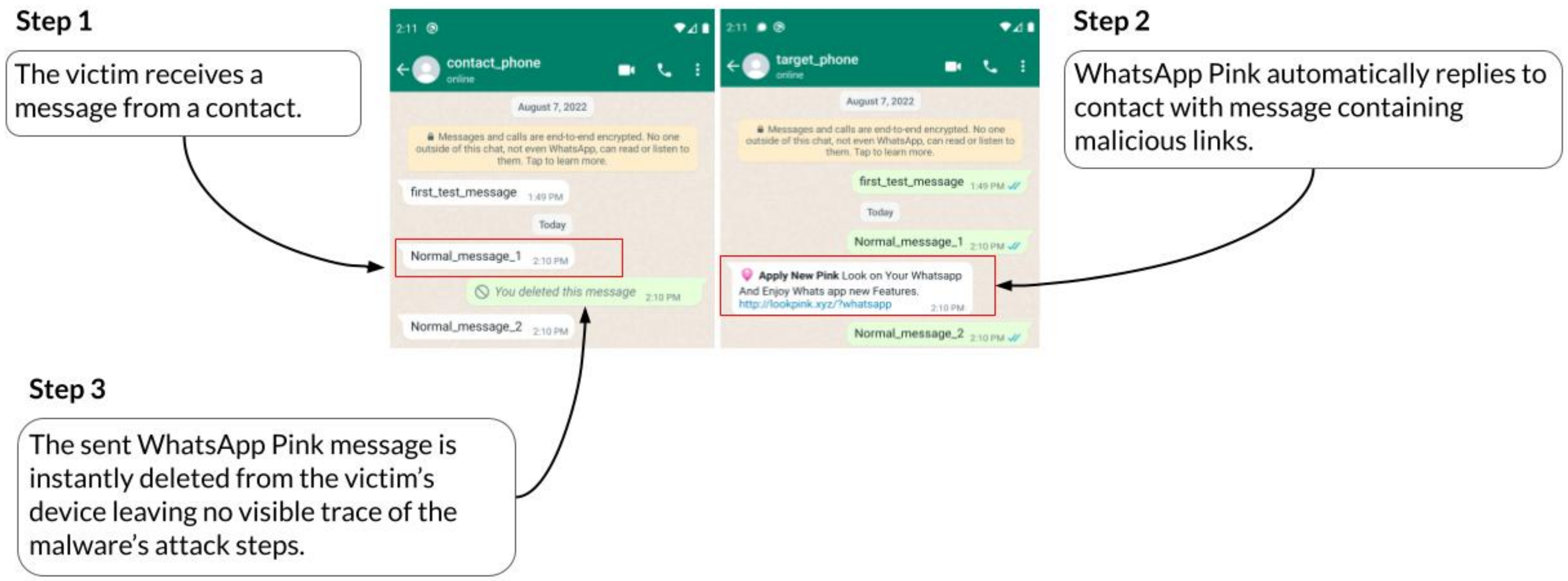
Model	OS	Chipset	CPU	RAM	Storage
Nexus 5 (low-end)	Android 6 (upgraded to Android 8)	Qualcomm MSM8992 Snapdragon 808 (20 nm)	Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57)	2GB	32GB
Google Pixel 4 (mid-range)	Android 10	Qualcomm SM8150 Snapdragon 855 (7 nm)	Octa-core (1x2.84 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.78 GHz Kryo 4	6GB	64GB
Samsung S21 (high-end)	Android 11	Snapdragon 888 / Exynos 2100	Octa-core (1x2.84 GHz Kryo 680 & 3x2.42 GHz Kryo 680 & 4x1.80 GHz Kryo 680) - USA/China	12GB	128GB

6.1.4 Experiment setup

MobFor is evaluated across three commercially available stock Android physical devices listed in Table 6.3, reflecting a range of device capabilities and limitations. Each of the devices has a JIT-MF enhanced version of WhatsApp v2.21.14.25 installed through MobFor and the additional JIT-MF Driver defined in Listing A.7 (in Appendix A). Regarding JIT-MF Driver development, WhatsApp is a privately-owned, closed-source app with protections against app repackaging. Therefore, an SQLite API-based JIT-MF Driver (whose accuracy and widespread use across messaging apps has also been previously demonstrated (Section 5.4)) was used to alleviate extensive compiled obfuscated code analysis that would have otherwise been required to select *Evidence_object*s and *Trigger_point*s. In this case, some compiled code analysis was still required to mitigate the app repackaging checks, which prevented the app from being enhanced with a JIT-MF Driver.

This experiment is carried out three times, once for each device available. Each experiment run involves using two devices: i) target device (D); and ii) contact device (C). The stealthy WhatsApp Pink malware is installed and set up each time on the target device (D) while contact phone (C) emulates communication via WhatsApp to and from device D. Figure 6.6 (reproduced from Chapter 3) shows how a conversation initiated by device C, is then simulated by both devices using `AndroidViewClient` to send 100, fifteen character-long, pseudo-random messages back and forth across devices D and C. Since device C initiates the conversation, the stealthy WhatsApp Pink malware installed on target device D immediately propagates the malicious link. It then deletes the message on device D. The malware is then uninstalled.

Late detection is simulated by a thirty-minute wait, at the end of which all *JIT-MF_Logs* containing memory dumps are collected from the device, along with the additional sources highlighted in Section 6.1.3. The available forensic tools (including MobFor) are used as described in Table 6.2. Finally, the parsed forensic artefacts populate a forensic timeline using Timesketch.


Step 1

The victim receives a message from a contact.

2:11

contact_phone
online

August 7, 2022

Messages and calls are end-to-end encrypted. No one outside of this chat, not even WhatsApp, can read or listen to them. Tap to learn more.

first_test_message 1:49 PM

Today

Normal_message_1 2:10 PM

You deleted this message 2:10 PM

Normal_message_2 2:10 PM

2:11

target_phone
online

August 7, 2022

Messages and calls are end-to-end encrypted. No one outside of this chat, not even WhatsApp, can read or listen to them. Tap to learn more.

first_test_message 1:49 PM

Today

Normal_message_1 2:10 PM

Apply New Pink Look on Your Whatsapp And Enjoy Whats app new Features.
http://lookpink.xyz/?whatsapp 2:10 PM

Normal_message_2 2:10 PM

Step 2

WhatsApp Pink automatically replies to contact with message containing malicious links.

Step 3

The sent WhatsApp Pink message is instantly deleted from the victim's device leaving no visible trace of the malware's attack steps.

Figure 6.6: Enhanced WhatsApp Pink attack steps (reproduced from Chapter 3).

Table 6.4: Incident Response tools artefact recovery.

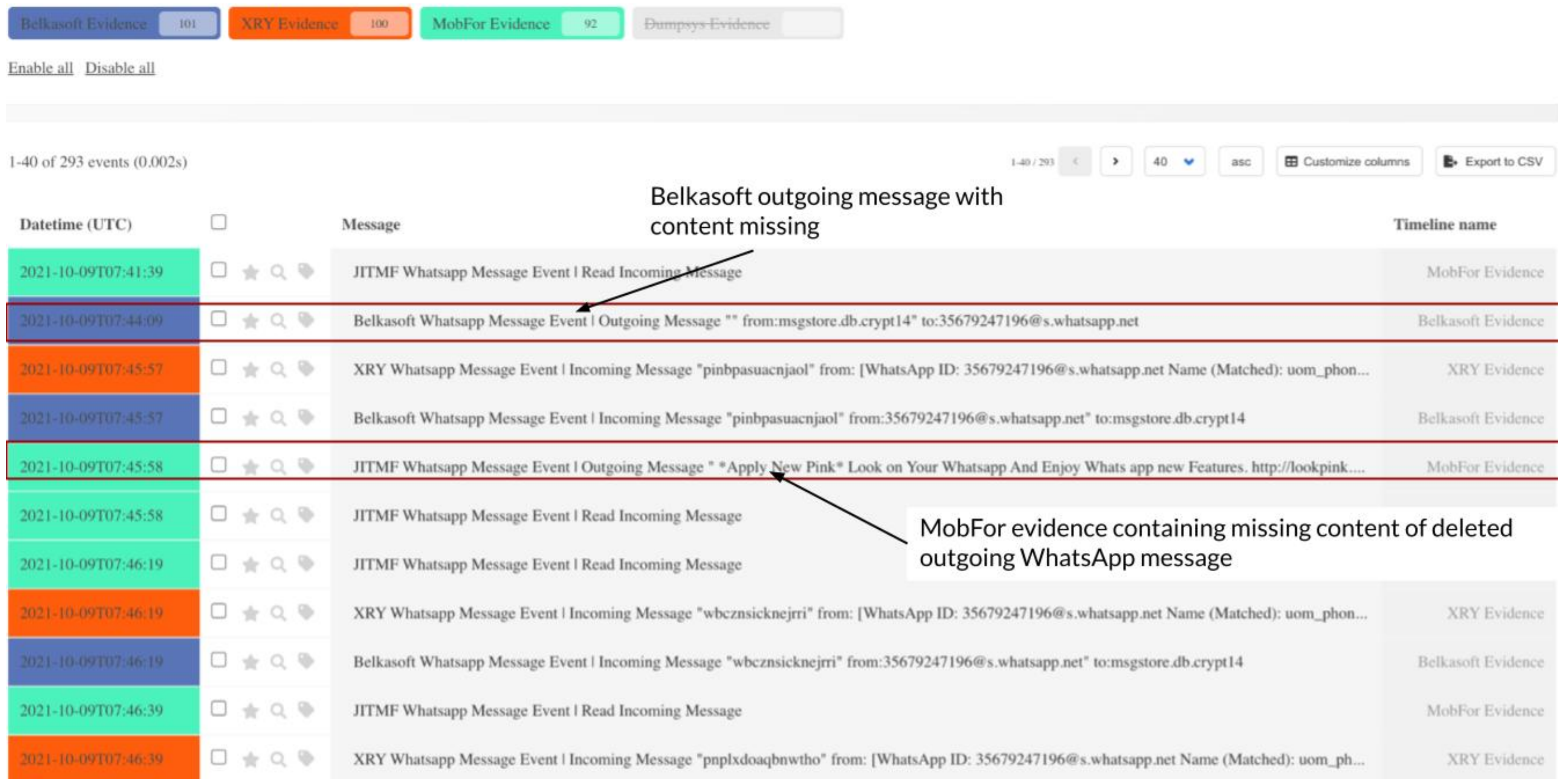
Key Attack Steps Metadata Recovered	Forensic Configuration								
	Nexus 5			Google Pixel 4			Samsung S21		
	Belkasoft	XRY	MobFor	Belkasoft	XRY	MobFor	Belkasoft	XRY	MobFor
Message Content	**-**	**-**	**X**	**-**	**-**	**X**	**-**	**-**	**X**
Message Sent Event	X	-	X	X	-	X	X	-	X
Message Deleted Event	-	-	-	-	-	-	-	-	-
Message Recipient	X	-	X	X	-	X	X	-	X
Message Sender	X	-	X	X	-	X	X	-	X
Message Timestamp	X	-	X	X	-	X	X	-	X

6.1.5 MobFor real-world scenario: Results

While Belkasoft and XRY gathered several artefacts, the results presented here focus on those related to WhatsApp messaging, given that the incident scenario in question concerns this messaging app.

Table 6.4 describes the findings of each forensic analysis configuration and device, concerning the known attack steps executed (the ground truth). While none of the forensic tools' outputs explicitly indicated that a WhatsApp message was stealthily deleted, critical key attack steps were still recovered. Crucially, MobFor was the only forensic tool out of the three that was able to recover the contents of the deleted message as shown in Figure 6.7 and Figure 6.8. Belkasoft was able to retrieve the event of a message being sent. However, the message content was missing. While this event is suspicious, without the message content, it is unclear whether or not this was: a simple message deleted by the target victim, a message with no content, or a malicious message propagated by the malicious app. XRY's output does not show the message-sending event at all. It is probable that since the content was missing in the WhatsApp databases, the event was not even displayed to the investigator to start with due to deletion.

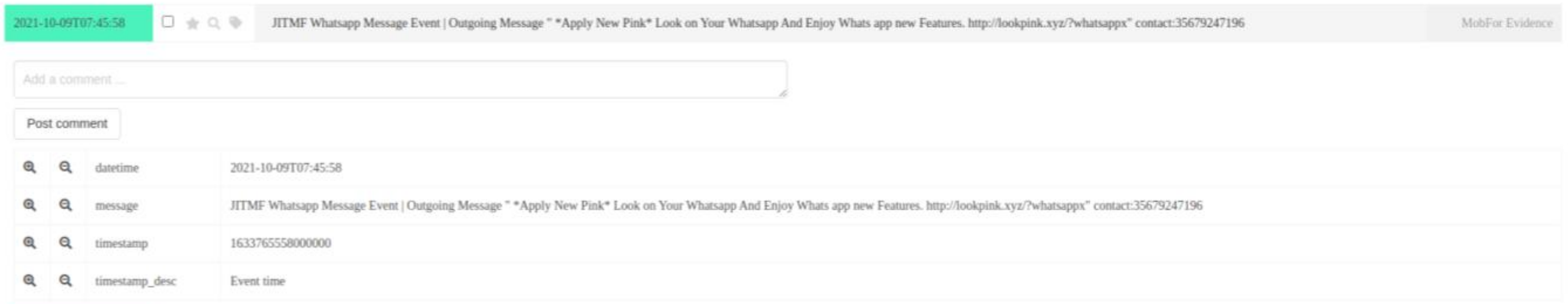
Forensic analysis. Timesketch is used to create a unified version timeline, including all WhatsApp evidence produced by the three forensic tools used. Since neither of the forensic tools used parses `dumpsys` logs, another parser was created for this purpose. Unlike the output generated by the three forensic tools described in Table 6.2, `dumpsys` provides information about system services rather than that related to a single app and therefore produces more events that tend to be noisier and may not be directly related to the incident that is under investigation. In this case, the critical attack steps involve a single message sent once to a single contact. Due to the minimal activity produced by the malware, as is typical with stealthy malware campaigns, it is difficult to highlight which



1-40 of 293 events (0.002s)

Datetime (UTC)	Message	Timeline name
2021-10-09T07:41:39	JITMF Whatsapp Message Event I Read Incoming Message	MobFor Evidence
2021-10-09T07:44:09	Belkasoft Whatsapp Message Event I Outgoing Message "" from:msgstore.db.crypt14" to:35679247196@s.whatsapp.net	Belkasoft Evidence
2021-10-09T07:45:57	XRY Whatsapp Message Event I Incoming Message "pinbpasuacnjaol" from: [WhatsApp ID: 35679247196@s.whatsapp.net Name (Matched): uom_phon...	XRY Evidence
2021-10-09T07:45:57	Belkasoft Whatsapp Message Event I Incoming Message "pinbpasuacnjaol" from:35679247196@s.whatsapp.net" to:msgstore.db.crypt14	Belkasoft Evidence
2021-10-09T07:45:58	JITMF Whatsapp Message Event I Outgoing Message " *Apply New Pink* Look on Your Whatsapp And Enjoy Whats app new Features. http://lookpink....	MobFor Evidence
2021-10-09T07:45:58	JITMF Whatsapp Message Event I Read Incoming Message	
2021-10-09T07:46:19	JITMF Whatsapp Message Event I Read Incoming Message	
2021-10-09T07:46:19	XRY Whatsapp Message Event I Incoming Message "wbcznsicknejrri" from: [WhatsApp ID: 35679247196@s.whatsapp.net Name (Matched): uom_phon...	XRY Evidence
2021-10-09T07:46:19	Belkasoft Whatsapp Message Event I Incoming Message "wbcznsicknejrri" from:35679247196@s.whatsapp.net" to:msgstore.db.crypt14	Belkasoft Evidence
2021-10-09T07:46:39	JITMF Whatsapp Message Event I Read Incoming Message	MobFor Evidence
2021-10-09T07:46:39	XRY Whatsapp Message Event I Incoming Message "pnplxdoaqbnwtho" from: [WhatsApp ID: 35679247196@s.whatsapp.net Name (Matched): uom_ph...	XRY Evidence

Figure 6.7: Identification of a suspicious event due to differences between forensic sources outputs related to the same event.


2021-10-09T07:45:58 JITMF Whatsapp Message Event | Outgoing Message " *Apply New Pink* Look on Your Whatsapp And Enjoy Whats app new Features. http://lookpink.xyz/?whatsappx" contact:35679247196 MobFor Evidence

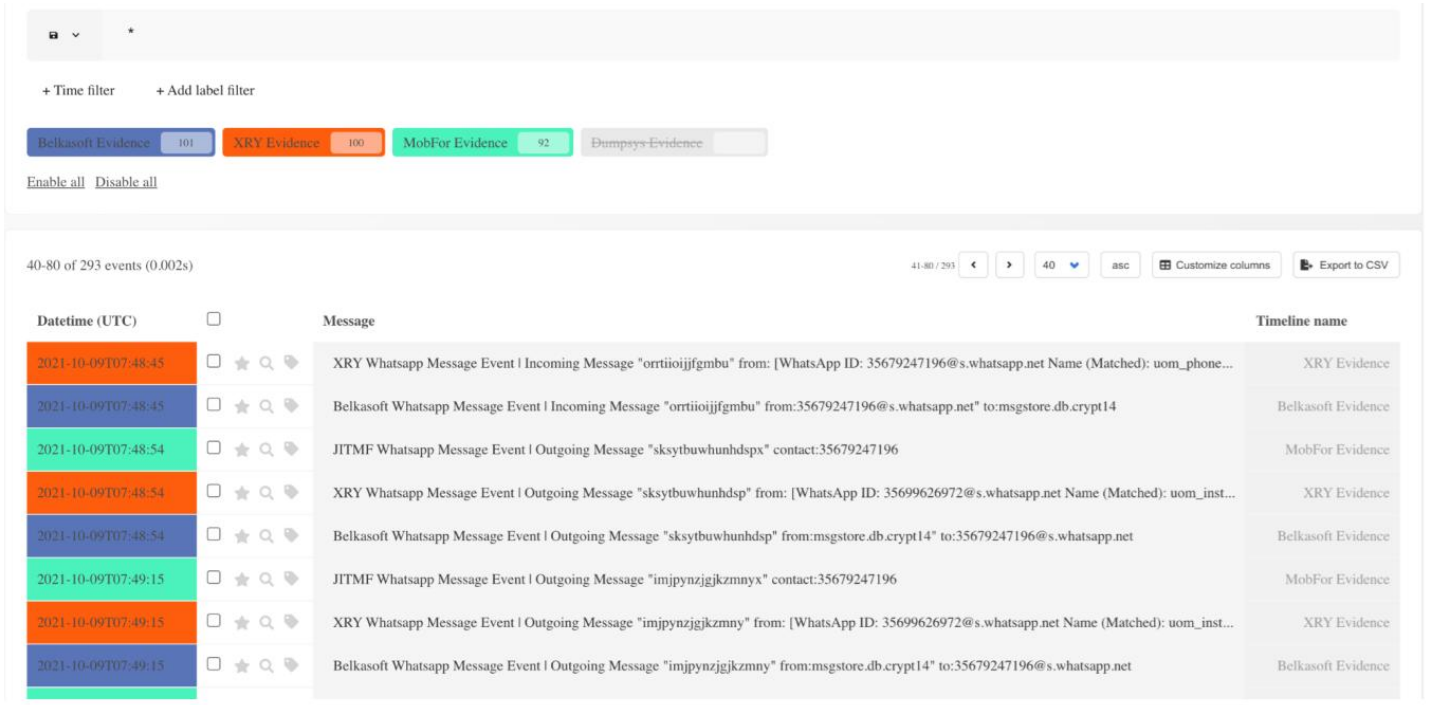
datetime	2021-10-09T07:45:58
message	JITMF Whatsapp Message Event | Outgoing Message " *Apply New Pink* Look on Your Whatsapp And Enjoy Whats app new Features. http://lookpink.xyz/?whatsappx" contact:35679247196
timestamp	1633765558000000
timestamp_desc	Event time

Figure 6.8: Additional metadata produced by MobFor in the case of a suspicious messaging event.

events in a timeline should be considered suspicious, especially amidst a substantial amount of evidence that is gathered. In this case, on average, 1,598 events were generated for the duration of the experiment ($\sim$ thirty minutes), of which only three events were related to the malware's activities. Therefore, the investigation focuses solely on the events produced by forensic tools in Table 6.2.

Figure 6.9 shows the evidence produced by the three sources when a *typical* WhatsApp message is *sent*. All three sources can produce the following: i) The event itself and ii) The metadata of the event, including the message content. This pattern is common in all messages sent between devices D and C, except for one event. Specifically, evidence for this event is missing in XRY and no metadata is available in Belkasoft. In contrast, MobFor output shows that the message content contains a link as shown in Figure 6.7. The discrepancy between the tools' outputs regarding the same event already suggests that this event is suspicious. This message is not visible on the target's device (Figure 6.6), which allows investigators to conclude that the message has been deleted from the target's device.

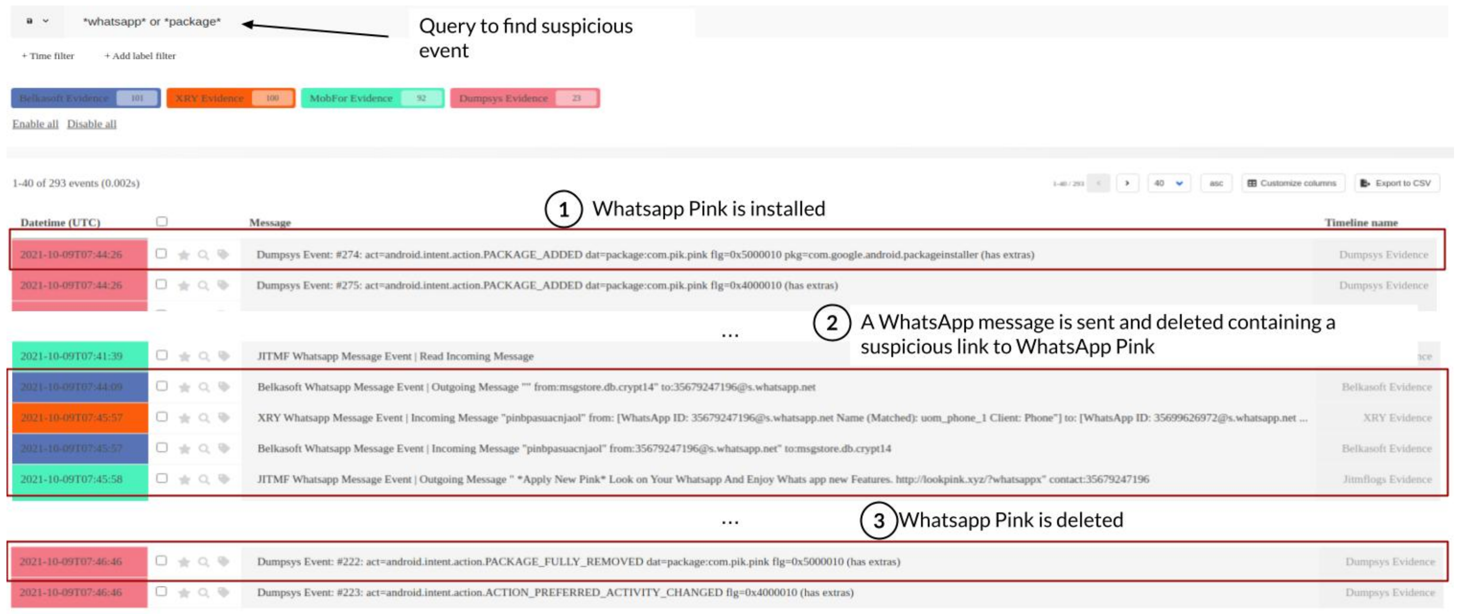
The investigation is broadened to include evidence from `dumpsys` and assess contents for potential attack steps. Due to this source's large number of events, the scope can be narrowed by creating a query to look for artefacts containing `whatsapp` data or `package`-related events. The steps the malware executes are retrieved as shown in Figure 6.10. The steps show the malware being installed, the presence of a message that has since been deleted, and the stealthy self-removal of a malicious app. This sequence of footprints is enough for the investigator to conclude that these steps were indeed carried out by malware on the victim's device. Crucially, while all forensic sources obtained were required to derive the exact and complete steps executed by the malware on the device, MobFor uniquely contributed to the forensic timeline, providing key elements whose presence steered the investigation in the right direction. Simultaneously, the results of this experiment demonstrated that further experimentation is required to determine how possibly hijacked attack steps can be better brought to the investigator's attention, in light of all the possible events collected to be forensically analysed.



40-80 of 293 events (0.002s)

Datetime (UTC)	Message	Timeline name
2021-10-09T07:48:45	XRY Whatsapp Message Event | Incoming Message "orrtiioijjfgmbu" from: [WhatsApp ID: 35679247196@s.whatsapp.net Name (Matched): uom_phone...	XRY Evidence
2021-10-09T07:48:45	Belkasoft Whatsapp Message Event | Incoming Message "orrtiioijjfgmbu" from:35679247196@s.whatsapp.net" to:msgstore.db.crypt14	Belkasoft Evidence
2021-10-09T07:48:54	JITMF Whatsapp Message Event | Outgoing Message "sksytbuwhunhdspx" contact:35679247196	MobFor Evidence
2021-10-09T07:48:54	XRY Whatsapp Message Event | Outgoing Message "sksytbuwhunhdsp" from: [WhatsApp ID: 35699626972@s.whatsapp.net Name (Matched): uom_inst...	XRY Evidence
2021-10-09T07:48:54	Belkasoft Whatsapp Message Event | Outgoing Message "sksytbuwhunhdsp" from:msgstore.db.crypt14" to:35679247196@s.whatsapp.net	Belkasoft Evidence
2021-10-09T07:49:15	JITMF Whatsapp Message Event | Outgoing Message "imjpynzjgjkzmnyx" contact:35679247196	MobFor Evidence
2021-10-09T07:49:15	XRY Whatsapp Message Event | Outgoing Message "imjpynzjgjkzmny" from: [WhatsApp ID: 35699626972@s.whatsapp.net Name (Matched): uom_inst...	XRY Evidence
2021-10-09T07:49:15	Belkasoft Whatsapp Message Event | Outgoing Message "imjpynzjgjkzmny" from:msgstore.db.crypt14" to:35679247196@s.whatsapp.net	Belkasoft Evidence

Figure 6.9: Combined timeline of events produced by Belkasoft, XRY and MobFor, in Timesketch.


Query to find suspicious event
1 Whatsapp Pink is installed
2 A WhatsApp message is sent and deleted containing a suspicious link to WhatsApp Pink
3 Whatsapp Pink is deleted
+ Time filter
+ Add label filter
Enable all
Disable all
Customize columns
Export to CSV

whatsapp or *package*

Belkasoft Evidence 101 | XRY Evidence 100 | MobFor Evidence 92 | Dumpsys Evidence 23

1-40 of 293 events (0.002s)

Datetime (UTC)	Message	Timeline name
2021-10-09T07:44:26	Dumpsys Event: #274: act=android.intent.action.PACKAGE_ADDED dat=package:com.pik.pink flg=0x5000010 pkg=com.google.android.packageinstaller (has extras)	Dumpsys Evidence
2021-10-09T07:44:26	Dumpsys Event: #275: act=android.intent.action.PACKAGE_ADDED dat=package:com.pik.pink flg=0x4000010 (has extras)	Dumpsys Evidence
…		
2021-10-09T07:41:39	JITMF Whatsapp Message Event | Read Incoming Message	
2021-10-09T07:44:09	Belkasoft Whatsapp Message Event | Outgoing Message "" from:msgstore.db.crypt14" to:35679247196@s.whatsapp.net	Belkasoft Evidence
2021-10-09T07:45:57	XRY Whatsapp Message Event | Incoming Message "pinbpasuacnjaol" from: [WhatsApp ID: 35679247196@s.whatsapp.net Name (Matched): uom_phone_1 Client: Phone"] to: [WhatsApp ID: 35699626972@s.whatsapp.net ...	XRY Evidence
2021-10-09T07:45:57	Belkasoft Whatsapp Message Event | Incoming Message "pinbpasuacnjaol" from:35679247196@s.whatsapp.net" to:msgstore.db.crypt14	Belkasoft Evidence
2021-10-09T07:45:58	JITMF Whatsapp Message Event | Outgoing Message " *Apply New Pink* Look on Your Whatsapp And Enjoy Whats app new Features. http://lookpink.xyz/?whatsappx" contact:35679247196	Jitmflogs Evidence
…		
2021-10-09T07:46:46	Dumpsys Event: #222: act=android.intent.action.PACKAGE_FULLY_REMOVED dat=package:com.pik.pink flg=0x5000010 (has extras)	Dumpsys Evidence
2021-10-09T07:46:46	Dumpsys Event: #223: act=android.intent.action.ACTION_PREFERRED_ACTIVITY_CHANGED flg=0x4000010 (has extras)	Dumpsys Evidence

Figure 6.10: Recovered sequence of attack steps from Timesketch timeline.

6.2 *JIT-MF_Logs* for anomaly detection and malware entry point identification

Results from the previous section demonstrate how *JIT-MF_Logs* contain unique evidence that complements mobile forensic tools in the case of a real-world app hijack case study. Crucially, *JIT-MF_Logs* have been shown to contain evidence that is critical to investigating app hijack threats, which other forensic tools do not collect. Yet, using an app-invasive approach (app repackaging) for JIT-MF installation that conflicts with app protection measures complicates matters. Moreover, applying *JIT-MF_Logs* for forensic analysis requires further experimentation to demonstrate its potential value in uniquely collecting evidence of app hijack and enabling anomaly detection resulting in precise (Section 5.3) and full attack step reconstruction (up to malware entry point).

The experiment described in this section aims to demonstrate how *JIT-MF_Logs* can be used in a SOC (Security Operations Center) setup, whereby forensic analysis on logs enables the detection and malware entry point identification of app hijack attacks. In this setup, JIT-MF compliments a mobile EDR, which collects all possible evidence from the device in real-time. Furthermore, for the first time, app-level virtualisation was used in this experiment to enhance the target app, rather than app repackaging. Thus opting for a non-invasive installation of JIT-MF and, therefore, the most feasible and compatible JIT-MF installation approach among apps. An additional detection component, sensitive to *JIT-MF_Logs*, was created for anomaly detection. To demonstrate an automated forensic analysis process involving *JIT-MF_Logs*, a range of IM hijack attack case studies were simulated based on realistic WhatsApp Pink worm propagation techniques.

6.2.1 VEDRANDO: Anomaly detection using *JIT-MF_Logs*

VEDRANDO (Volatile-memory-enhanced EDR for ANDrOid) is an enhanced EDR that was developed for this experiment. Figure 6.11 gives an overview of its architecture, which consists of two main components: the *Events Collector* and the *Attack Detector*. The *Events Collector* collects evidence from the device. The *Attack Detector* detects and reconstructs attack steps of stealthy app hijack attacks through a detection methodology that applies standard anomaly detection and correlation techniques.

6.2.1.1 Events Collector

The *Events Collector* component in Figure 6.11 illustrates a high-level view of the EDR setup comprising of the following components: an EDR server, an EDR client (mobile

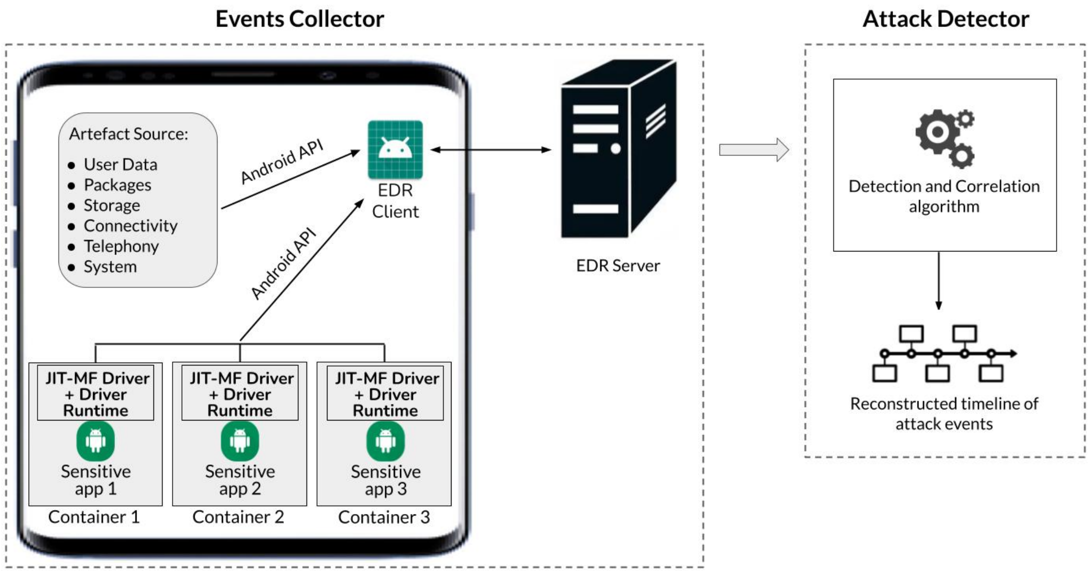


Figure 6.11: Overview of VEDRANDO architecture.

app) and trusted app-level virtualisation containers that each host an app which may be targeted in an app hijack attack. Each container contains an additional library to serve as the JIT-MF Driver Runtime and JIT-MF Drivers. The trusted app-level virtualisation containers are implemented to load the JIT-MF Driver and Runtime when the targeted app is launched. This enables the operation of JIT-MF to timely collect artefacts from the plugin app memory without requiring app repackaging.

While the make-up of each container is the same, different sensitive apps are hosted in different containers to maintain the application sandbox protections that Android offers between apps out-of-the-box.

Implementation. The *Events Collector* component of VEDRANDO was implemented by extending ReLF [152]. ReLF is the only open-source EDR tool available for mobile phones, with the ability to collect critical evidence of sensitive app events found in memory produced by JIT-MF Drivers. ReLF extends GRR [35], an open-source, scalable system developed by Google for remote live forensics and incident response. It enables forensic investigations of Android devices by acquiring various forensic artefacts from devices, as many as any other such forensic tools, as shown in Table 6.5. As with typical EDRs, the setup involves having ReLF clients on mobile phones, from which events will be collected and sent to a ReLF server. The ReLF client may be built and deployed as a user or system app. The latter has access to more forensic sources (see sources marked with $*$ in Table 6.5) but requires root access. ReLF client apps built as user apps

Table 6.5: Forensic artefacts sources collection comparison of Android Forensic EDR tools, as shown in [152], as of 2021. Commercial tools in this study included: Oxygen Forensic Suite, MOBILedit, XRY Forensic Examiner's Kit, UFED and EnCase Mobile Investigator

	Artefact source	***SystemSens** [47]*	***DroidWatch** [55]*	***Device Analyzer** [134]*	***AFLogical OSE** [1]*	***DELTA** [112]*	***Andriller** [106]*	***Commercial Tools***	***ReLF** [152]*
System	OS info			✓			∗	∗	✓
	Hardware info			✓			∗	∗	✓
	System settings			✓			✓	∗	✓
	Battery statistic	✓		✓		✓		∗	✓
App	Installed packages		✓	✓		✓	✓	∗	✓
	Running processes	✓		✓		✓		∗	∗
Telephony	Contacts		✓	✓	✓		∗	∗	✓
	Call logs	✓	✓	✓	✓	✓	∗	∗	✓
	SMS/MMS	✓	✓	✓	✓	✓	∗	∗	✓
	SIMs & subscriptions			✓			∗	∗	✓
	Cellular info	✓		✓		✓		∗	✓
Connectivity	Wi-Fi status & hotspots	✓		✓		✓	✓	∗	✓
	Bluetooth info			✓		✓		∗	✓
	NFC status							∗	✓
	VPN profiles								∗
	NIC info & Netstat	✓		✓				∗	∗
Storage	Storage volume info			✓		✓		∗	✓
	Filesystem & file stats			✓		✓		∗	∗
	Retrieve arbitrary file			✓		✓		∗	∗
Sensors	Location		✓	✓		✓		∗	✓
	Microphone					✓		∗	
	Sensor info & logging			✓		✓		∗	✓
User Data	User accounts		✓				∗	∗	✓
	Device user profiles					∗		∗	∗
	Calender	✓	✓				∗	∗	
	Browser history		✓				∗	∗	∗
Other	Screen state & capture	✓	✓	✓		∗	∗	∗	
	Key & touch logging					∗			
	Remote logging	✓	✓	✓		✓			✓

✓ refer to forensic artefacts that can be collected by the EDR tool.
∗ refer to forensic artefacts that can be collected *only* if the EDR tool has system privileges (`adb` or `root`) on the device.

interact with the underlying system through Android APIs or the low-level ReLF native service using Inter-Process Communication (IPC) [152]. JIT-MF Drivers in different containers may be the same if the sensitive apps (1,2 and 3 in Figure 6.11) use a common infrastructure (which evaluation results demonstrate this is very likely the case). In this case, the EDR client and server are the ReLF client app and server, respectively.

While the app is in use, *JIT-MF_Logs* are populated at runtime, based on *Evidence_object*s and *Trigger_point*s defined in the JIT-MF Driver located in the container. When an alarm is raised, the ReLF server can invoke artefact collection flows, instructing the ReLF client to collect any pending logs not yet collected through continuous monitoring to send back to the server as part of evidence collection to aid the ongoing investigation. As shown in Figure 6.11, the ReLF client leverages the Android API to collect all Android forensic sources, including logs containing in-memory evidence collected by the JIT-MF Driver deployed within VirtualApp [131]. For *JIT-MF_Logs* generated by JIT-MF Drivers containing evidence from app memory, the client uses the Android API to search for files on the device with a `*.jitmflog` extension.

6.2.1.2 Attack Detector

Anomaly detection of logs is commonly used to detect anomalous behaviour. In the case of stealthy app hijack attacks, existing log sources, such as third-party application logs, do not provide enough evidence to enable anomaly detectors to detect a specific app event as anomalous. The additional *JIT-MF_Logs* containing evidence from memory collected by VEDRANDO's *Events Collector* provide the necessary additional context with which standard anomaly detection methods can observe a difference between normal app usage and app hijack, thus enabling the detection of anomalous events as hijacked benign app events, even in the case of stealthy attacks. While evidence of attack steps from hijacked apps can be found in logs produced by the *Events Collector* component, stealthy attacks may consist of several steps whose footprints are dispersed across many separate logs on different victims' devices.

The *Attack Detector* component of VEDRANDO comprises the detection algorithm outlined in Algorithm 2. The algorithm uses an existing, standard anomaly detection method to detect anomalies in the *JIT-MF_Logs*, then correlates anomalies with events collected from other logs found on the device to reconstruct all the attack steps, including malware entry point. The algorithm takes as input logs produced by the *Events Collector* component, comprising *JIT-MF_Logs* with evidence objects from app memory and other logs found on the device (see Table 6.5), and a user-defined configuration *Config c*. The configuration variable *Config c* holds settings for generating the anomaly detection model. Namely, it comprises: i) the anomaly detection method (a); ii) associated features selected ($[f_1 ... f_n]$); iii) the anomaly threshold value t, which will be used to identify data points as anomalous; and iv) a list of app-specific regex keywords ($[p_1 ... p_n]$) used during the correlation of events. The algorithm outputs a list of events *Correlated_Events e*, attributed to the complete attack steps.

Anomaly detection. All the entries from different log sources are parsed (*line 1* in Algorithm 2), so each entry has three main fields: i) timestamp, ii) log source, and iii) activity. *JIT-MF_Logs* are filtered to remove duplicates. Furthermore, in the case of the third-party app and *JIT-MF_Logs*, logs are filtered so that only sources of evidence related

Algorithm 2: Anomaly detection and correlation algorithm

```
Input  : JITMF Logs $J$, Other Logs $O$, Config $c$= {Anomaly Detection Method $a$,
         Anomaly Detection Features $[f_1...f_n]$, Anomaly Detection Threshold $t$,
         Correlation Keywords Regex $[p_1...p_n]$}
Output: Correlated_Events e=∅

1  $J, O \leftarrow ParseLogs(J, O)$
2  $J, O \leftarrow GetAnomalies(J, O, c)$
3
4  function GETANOMALIES (JITMF Logs J, Other Logs O, Config c):
5      $m \leftarrow GenerateModel(O, J, c[a], c[[f_1...f_n]])$
6      $jitmf\_anomalous\_log\_entries \leftarrow DetectAnomalies(J, m, c[t])$
7      $e \leftarrow Correlate(jitmf\_anomalous\_log\_entries, J, O, c[[p_1...p_n]])$
8      return e
9
10 function CORRELATE (jitmf_anomalous_log_entries, JITMF Logs J, Other Logs O,
    Correlation Keywords Regex $[p_1...p_n]$):
11     Events e=∅
12     foreach $an \in jitmf\_anomalous\_log\_entries$ do
13         if $IsTimestamp(an)$ then
14             $jitmf\_log\_entry \leftarrow GetJITMFLogEntryAt(an)$
15         else
16             $jitmf\_log\_entry \leftarrow an$
17         end
18         $obj \leftarrow GetEvidenceObject(jitmf\_log\_entry)$
19
20         $/ * Feature-based correlation * /$
21         foreach $p_i \in [p_1...p_n]$ do
22             if $obj.match(p_i)$ then
23                 $keyword \leftarrow obj[p_i]$
24                 $e \leftarrow e \bigcup FindEventsWithKeyword(O, J, p_i)$
25             end
26         end
27     end
28
29     $/ * Time-based correlation * /$
30     $jitmf\_anomalous\_logs \leftarrow SortByTime(jitmf\_anomalous\_log\_entries)$
31     $start\_time \leftarrow GetFirstEventTime(jitmf\_anomalous\_logs)$
32     $end\_time \leftarrow GetLastEventTime(jitmf\_anomalous\_logs)$
33     $e \leftarrow e \bigcup GetEventsInTime(O, start\_time, end\_time)$
34     return e
```

to the evidence object are considered. For instance, in a messaging hijack attack, where the evidence object is a message sent from the user's phone, the evidence collected from the app is its database comprising many tables possibly containing data unrelated to messaging, e.g. app themes, which may cloud the investigation. The function *GetAnomalies*() is then called with the following parameters: i) parsed *JIT-MF_Logs* (*J*; ii) logs from other sources (*O*); and iii) configuration settings (*Config c*).

The function *GetAnomalies*() first generates a machine learning anomaly detection model based on the machine learning method and features defined in the user-inputted configuration (*line 5*). The model *m* is then applied on the parsed and filtered set of *JIT-MF_Logs J* using the threshold defined in the configuration settings (*line 6*). The anomalous JIT-MF log entries revealed by the anomaly detection model are considered anomalous JIT-MF events. These are then correlated (*line 7*) with other log events (*JIT-MF_Logs* and logs from other sources) using the app-specific correlation regex keywords given parameter ($[p_1...p_n]$). The function returns the result of the *Correlate*() function.

Correlation. Individual entries in *JIT-MF_Logs* include a timestamp and metadata of the *Evidence_object* definition as described in the JIT-MF Driver. Regardless of the driver implementation or the app, the contents of the *Evidence_object* can be parsed to derive relevant keywords used during an attack step. App-specific correlation keyword regexes retrieve the relevant keywords from anomalous JIT-MF log entries. While an API-based JIT-MF Driver means that the same *Evidence_object* can be used across different apps that use the same API, the make-up of the *Evidence_object* is still app-specific. Therefore, the regex pattern used to retrieve this metadata or identifier from a log entry must also be app-specific. That said, there are cases where keyword regex is the same across apps due to formatting standards e.g., object ID format may be in UUID format, which is a standard format.

The *Correlate*() function accepts as input the anomalies detected, *JIT-MF_Logs*, logs from other sources and the app-specific correlation regex keywords. If the anomaly is a timestamp (as is the case with time-based anomaly detection), *JIT-MF_Logs* at that time are retrieved (*lines 11-17* in Algorithm 2). The *Evidence_object* of the anomalous JIT-MF log entry is retrieved (*lines 18*) and used to perform correlation as follows. The algorithm correlates anomalous JIT-MF log events with events in other log sources based on two mechanisms: i) *Feature-based correlation* and ii) *Time-based correlation*. It is unlikely that normal events have identical keywords in their *Evidence_object*. However, in the case of malware, especially during propagation, *Evidence_object*s containing matching keywords is expected. Therefore events in other log sources which contain the identical keywords as those found in anomalous JIT-MF log events (*lines 21-26*) are considered further attack steps, and are added to the list of *Correlated_Events e*. This is referred to as *Feature-based correlation*. Any other attack steps performed by the attack are assumed to have happened in the period within which the correlated list of attack steps occurred. Therefore *Time-based correlation* is used to search for other events that occur in other logs when JIT-MF log anomalies are detected (*lines 30-33*). This ensures any attack steps carried outside the app functionality are also disclosed. Any log entries found through correlation are entered into a set of correlated events and returned to the analyst or investigator as the complete list of attack steps carried out.

6.2.2 Setup performance overheads

VEDRANDO's *Events Collector* set-up requires apps to be run in a virtual environment (through app-level virtualisation), having a JIT-MF Driver and Runtime installed. Using app-level virtualisation instead of app repackaging calls for a non-invasive approach to JIT-MF installation. Therefore, this increases the potential compatibility of JIT-MF installation across multiple Android apps. Yet this introduces additional performance costs due to the additional proxying done by the virtualisation layer.

The performance overheads of JIT-MF installed in a virtualised environment are calculated, based on the implementation of VEDRANDO. A set of apps from the most popular 100 apps in Google PlayStore in February 2022 (as listed on AppBrain), which were not previously installed on the phone or manufacturer-specific, are selected (a total of 33 apps).

Experiment setup. A stock (unrooted) Google Pixel 3a physical phone, with eight processors and 4GB RAM, was used, which runs on arm64-v8a CPU architecture and Android version 9 (as required by VirtualApp). The apps selected were downloaded from APKPure[4] using *apkeep*[5] to ensure that the APKs downloaded complied with the architecture and Android version. App activity was simulated using the *UI Exerciser Monkey* tool,[6] by injecting 20 random UI events with a throttle of 30s. This time buffer allows the virtual environment to spawn the app but cannot be reset to execute the rest of the events due to limitations of *UI Exerciser Monkey*. A seed value was used to ensure that the same app events could be repeated due to multiple runs.

The 33 apps were installed and used directly on the device to check typical resource usage. The apps ran in a standard VirtualApp container to evaluate their compatibility

[4]`https://apkpure.com/`
[5]`https://github.com/EFForg/apkeep`
[6]`https://developer.android.com/studio/test/other-testing-tools/monkey`

with the virtual environment. In this step, the overhead introduced by the Android virtualisation regarding CPU and memory usage were measured. At the end of this phase, five apps were identified as having triggered an exception due to incompatibility with the virtual environment. These apps were discarded from the rest of the experiments. The remaining 28 apps were executed three times i) directly on the device, ii) inside a simple VirtualApp container and iii) inside a VirtualApp container equipped with a generic SQLite API-based JIT-MF Driver (as implemented in VEDRANDO's *Events Collector*). The results were averaged over ten runs.

Results. Table 6.6 shows the minimum, average, and maximum overhead values expressed in percentage points (pp). In the first column, the execution of apps in a plain VirtualApp is compared to the traditional execution method (no virtualisation). In the second column, overheads introduced by the VirtualApp container as implemented in VEDRANDO's *Events Collector* component (i.e. with JIT-MF Driver and Runtime installed) is compared to the execution in a plain VirtualApp environment. Since the execution of an app under virtualisation is composed of two processes (the container and plugin), the overall amount of CPU and memory is given by the sum of the overhead of these two processes.

Table 6.6: Overall CPU and Memory usage overheads in percentage points (pp) of 28 most popular apps, when executing within VirtualApp and a JIT-MF-enhanced version of VirtualApp, respectively.

		VirtualApp *(added pp overheads on device)*	**VirtualApp with JIT-MF** *(added pp overheads on plain VirtualApp)*
CPU	min.	-0.46	+1.2
	avg.	+2.83	+2.06
	max.	+2.76	+6.79
Memory	min.	-0.48	+0.23
	avg.	+0.56	+0.18
	max.	+1.19	+0.29

Results from Table 6.6 show that when introducing virtualisation through VirtualApp, there is an average increase of 2.83pp in CPU usage and 0.56pp in memory usage. The results for the container as implemented in the *Events Collector* component, *using an SQLite API-based JIT-MF Driver* (shown to be effective among instant messaging apps Section 5.4), show that the additional average overhead introduced is negligible, i.e., an increase of 2.06pp for the CPU usage and 0.18pp for the memory. This increase is caused by the overhead required by JIT-MF Drivers to capture memory dumps on trigger points done through instrumenting methods. The overall additional CPU usage incurred by VEDRANDO's *Events Collector* component *when using SQLite API-based JIT-MF Drivers*, is on average 4.89pp, rendering it feasible in terms of runtime performance in a real-world scenario. This, however, may vary depending on the type of JIT-MF Driver used in the VirtualApp container.

6.2.3 Attack investigation case studies

VEDRANDO's *Attack Detector* is evaluated by measuring its ability to reveal attack steps related to stealthy app hijack attacks in a realistic scenario. Rather than assessing the performance of existing anomaly detection models on a large dataset, these case studies aim to demonstrate how anomaly detection methods typically available to SOC analysts through their SIEM (Security information and event management[7]) setup, can be used to detect anomalous events related to app hijack attacks when provided with *JIT-MF_Logs* produced by VEDRANDO's *Events Collector* component. To this end, similarly to other event reconstruction-related work [89, 6, 145], a qualitative case study is conducted. The study focuses on instant messaging (IM) hijack attacks and simulates the WhatsApp Pink attack to target Android's ten popular IM apps shown in Table 6.7.

Table 6.7: List of applications used in the case study.

App #	App Name	Package	Version	# of Downloads
1	Facebook	com.facebook.orca	392.0.0.12.106	5B+
2	WhatsApp	com.whatsapp	2.23.2.4	5B+
3	Imo	com.imo.android.imoim	2023.01.1031	1B+
4	Skype	com.skype.raider	8.92.0.401	1B+
5	Telegram	org.telegram.messenger.web	9.3.2	1B+
6	WhatsApp Business	com.whatsapp.w4b	2.23.5.77	500M+
7	Kik	kik.android	15.49.0.27501	100M+
8	Signal	org.thoughtcrime.securesms	6.12.5	100M+
9	Plus Messenger	org.telegram.plus	9.4.9.0	50M+
10	Slack	com.Slack	23.01.40.0	10M+

6.2.3.1 Case study setup

Figure 6.12 shows the experiment setup and flow, comprising an implementation of the working prototype for VEDRANDO, shown previously in Figure 6.11, and the inves-

[7]https://www.splunk.com/en_us/data-insider/what-is-siem.html

Table 6.8: Ground truth timeline of events for IM hijack case study.

Event	Event Description	Comments
❶	Authorised messages	Normal traffic consisting of outgoing messages using the app, occurring at a random time offset.
❷	Malware entry point	An incoming message via the app that contains a link to a malicious app: *DHL: Your parcel is arriving, track here: <URL>*
❸	Malicious *demo.apk* installed	The user clicks on the link which automatically downloads and installs the malicious app (*demo.apk*) silently.
❹	Link propagated to contacts in messaging app	*demo.apk* propagates itself by sending the same message with the malicious link to all the contacts available in the app.
❺	Propagated messages deleted	*demo.apk* deletes the sent messages from the victim's app.
❻	Trigger event	After one hour, a recipient of the message containing malicious content, alerts the victim that suspicious activity is occurring on their phone: *"Hey, I think something is wrong with your phone. You sent me a suspicious message."*.

tigation flow indicated by arrows. A stock (unrooted) Google Pixel 3a physical phone was used on which an implementation of VEDRANDO's *Events Collector* component, was deployed, using an SQLite API-based JIT-MF Driver,[8] whose accuracy has been demonstrated in previous experiments (Chapter 5). Normal traffic on each app consisted of loading and sending instant messages. This was simulated using *AndroidViewClient*,[9] assuming that the user messages random contacts from his list of contacts, waiting a random amount of seconds (between one and ten) before sending the message. While the simulation of normal traffic may threaten the validity of this experiment, the simulated traffic generated within the case study time window is a realistic enough representation to provide a basis for this study.

IM App Hijack attack simulation. Following the stealthy WhatsApp Pink attack outlined in Section 6.1, IM hijack attack scenarios misusing messaging functionality for propagation are simulated using `adb` shell commands and *AndroidViewClient*.[10]

Table 6.8 shows the ground truth timeline of events carried out to simulate the attack scenario for this case study. A malicious message is received containing a link to a malicious app (Table 6.8 ❷). Once the user clicks on the link, the app (a fake app called *demo.apk*) is silently installed (Table 6.8 ❸) and propagates to the user's contacts via the default IM app installed (in this case, the apps in Table 6.7). To attain stealth, the

[8]`https://gitlab.com/bellj/vedrando/-/tree/main/sqlite-jitmf-driver.js`
[9]`https://github.com/dtmilano/AndroidViewClient`
[10]`https://github.com/dtmilano/AndroidViewClient`

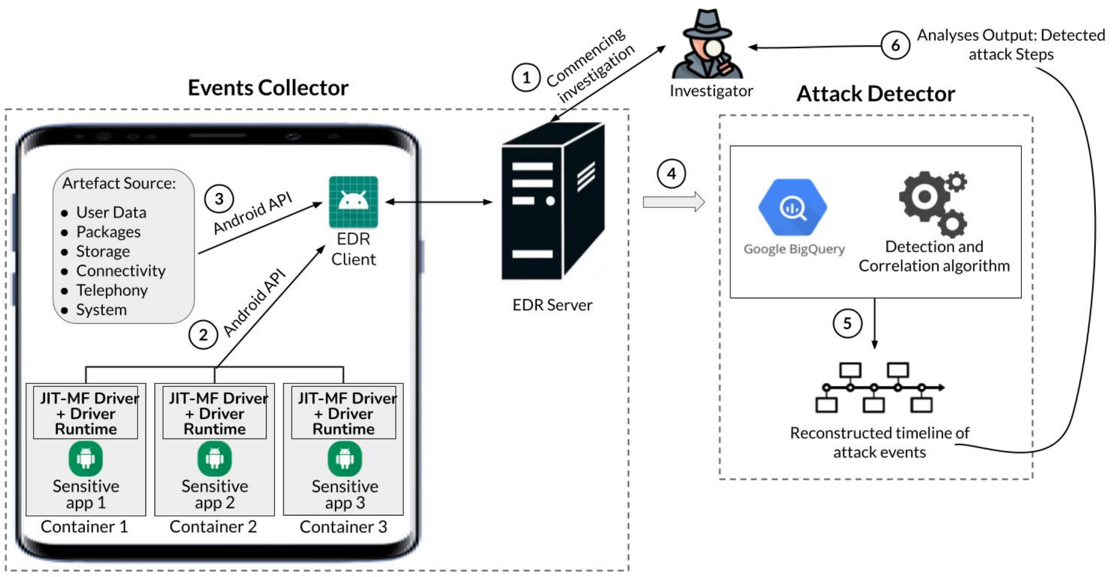


Figure 6.12: Complete case study experimentation flow.

simulated attack deletes the sent messages from the victim's phone (Table 6.8 ❺) and hides by removing the malicious app icon from the home screen so that the victim is unaware of the malicious app and it goes unnoticed by the victim for longer. ❻ in Table 6.8 is a *Trigger Event*; that is, it alerts the user that a suspicious event has possibly occurred, which initiates an investigation process. It is typical for realistic malware aiming to be stealthy to wait until it is *the right time* to execute [100]. In this case, the malware waits until the hijacked app is not in use so as not to alert the user of abnormal behaviour. Due to these stealthy measures and additional ones that the malware uses to hide its attack steps, the trigger event occurs much after (in this case, almost an hour later) the attack, meaning that the malware would have cleared its tracks leading to delayed detection.

Investigation setup. Assuming the role of an SOC analyst in an enterprise, an investigation process starts with the analyst who interacts with the GRR ReLF server (step 1 in Figure 6.12) to collect evidence artefacts from the victim's phone, including *JIT-MF_Logs* produced by the JIT-MF Driver (steps 2,3). SOC analysts are typically equipped with SIEM services that provide access to out-of-the-box anomaly detection tools. The GRR ReLF server has bindings to Google BigQuery,[11] a service that enables scalable analysis over petabytes of data and provides machine learning capabilities including anomaly detection. This service is used as a SIEM equivalent for this experiment. During the

[11]`https://cloud.google.com/bigquery`

investigation procedure followed in this evaluation, the artefacts collected by the ReLF client and sent to the GRR ReLF server are saved in Google BigQuery datasets (step 4). VEDRANDO's *Attack Detector* detection and correlation algorithm uses Google BigQuery's machine learning API to detect anomalies in the collected *JIT-MF_Logs*. These anomalies are correlated to events from other logs to reconstruct the attack steps executed by the messaging app hijack attack (step 5). The SOC analyst then analyses and interprets the attack steps detected (step 6).

6.2.3.2 Detection and correlation configuration

The investigation procedure outlined above was carried out after the attack hijack scenario was executed on each targeted messaging app shown in Table 6.7. Once the logs after each case study were retrieved, the detection and correlation algorithm (Algorithm 2) as part of VEDRANDO's *Attack Detector* component was implemented and executed using the configuration described below.

Anomaly detection models. Google BigQuery ML [54] provides anomaly detection capabilities through four machine learning model types: ARIMA_PLUS, K-means, PCA and Autoencoder. All these models are unsupervised and can therefore detect anomalies without needing labelled data. ARIMA_PLUS detects anomalies in time series data, while the others detect anomalies in independent and identically distributed random variables. For this evaluation, these models were used with selected applicable parameters and features as configuration input to the detection algorithm (Algorithm 2) to measure the algorithm's effectiveness in detecting and reconstructing attack steps. All collected logs (including *JIT-MF_Logs*) are processed in BigQuery, and preprocessed log content is used for building the different models. Hyperparameter tuning is commonly used to improve model performance by searching for optimal hyperparameters. During this evaluation, the default and recommended Vertex AI Vizier algorithm was used to tune hyperparameters.[12]

Dataset. The evaluation of machine learning algorithms typically involves using large, established datasets. However, this evaluation aims to demonstrate the value that *JIT-MF_Logs* bring to the incident response process by showing that evidence in these logs enables existing machine-learning anomaly detection models to detect anomalies in benign app activity related to app hijack that can help reconstruct attack steps. Therefore

[12] `https://cloud.google.com/bigquery/docs/reference/standard-sql/bigqueryml-hyperparameter-tuning`

the dataset used to train anomaly detection models in this evaluation is realistic to what an SOC would have available in such an incident. This comprises logs typically collected by EDRs (shown in Table 6.5) and *JIT-MF_Logs* that were populated during the case study (which involved both the attack and normal traffic) and collected as part of the investigation process by VEDRANDO's *Events Collector* component.

The VirtualApp container used by the *Event Collector* was built and deployed to the phone in debug mode, and therefore its app data could be retrieved. VirtualApp app data houses the data produced by plugin apps, and therefore relevant third-party app forensic sources could also be accessed and collected as forensic sources. When working with a VirtualApp container app that is not running in debug mode, forensic analysts can opt to use app features like 'Backup' or collaborate with the device owner to collect the evidence that is present in the app. Once the sources were collected, relevant data was extracted, related to the app's main functionality (in this case, messaging), converted into logs and transferred to the Google BigQuery dataset.

Evidence collected from the app (both *JIT-MF_Logs* and app-specific logs) comprised its database consisting of many tables possibly also containing data unrelated to messaging (e.g., app themes etc.), which do not contribute to the main app functionality. Therefore logs were filtered to include only evidence related to `messaging` activity. The timestamp, forensic source and activity fields of log entries for each source were identified, parsed and used to build anomaly detection models.

Features. Table 6.9 shows the features used per anomaly detection method to generate the anomaly detection models during the execution of Algorithm 2. Features were selected based on the anomaly detection method and knowledge that *JIT-MF_Logs* may contain evidence of offloaded attack steps that are not visible in other forensic sources. Log entries from multiple sources were parsed so that each had a timestamp, forensic source and activity. However, the format of the content inside the activity field differed from one forensic source to another, both across sources and in the case of app-specific logs and *JIT-MF_Logs*, even across apps. Rather than parsing each log type individually for each app and forensic source, derived features in the form of log entry amounts were used per feature grouped by a time window.
Feature 1 represents the discrepancy in log entry amount between that produced by all forensic sources (excluding *JIT-MF_Logs*) and the amount found in *JIT-MF_Logs*.
Feature 2 represents the total amount of log entries collected from all sources of the *Events Collector* component.
Features 3 represents the log entry amount collected from *JIT-MF_Logs*.
Features 4 - 9 represent the log entry amounts collected from distinct forensic sources

Table 6.9: The list of features used for anomaly detection model generation, as implemented in Algorithm 2. The time units used for each model generated are described in Table 6.10.

Method	**Feature**	**Description**
ARIMA_PLUS	Feature 1	Discrepancy between the amount of *JIT-MF_Logs* and other logs
K-Means, PCA, Autoencoder	Feature 2	Total amount of logs
	Feature 3	Amount of *JIT-MF_Logs*
	Features 4 - 9	Amount of logs per forensic source
	Feature 10	Amount of *JIT-MF_Logs* related to Data Retrieval (`SELECT`)
	Feature 11	Amount of *JIT-MF_Logs* related to Data Insertion (`INSERT`)
	Feature 12	Amount of *JIT-MF_Logs* related to Data Replacement (`REPLACE`)
	Feature 13	Amount of *JIT-MF_Logs* related to Data Update (`UPDATE`)
	Feature 14	Amount of *JIT-MF_Logs* related to Data Deletion (`DELETE`)

(excluding JIT-MF). While Table 6.5 shows that a typical collection involves retrieving multiple forensic sources, only five contained evidence during this case study.
Features 10-14 represent the log entry amounts collected from *JIT-MF_Logs* with distinct SQL statements. Since *JIT-MF_Logs* in these case studies are generated using an SQLite JIT-MF-based driver, log entries include SQL statements which process the message object (shown in Listing 6.1). The `SELECT`, `INSERT`, `REPLACE`, `UPDATE` and `DELETE` SQL statements are considered to reflect app functionality related to the processing of the message object. The amount of logs having a specific SQL statement is a feature.

```
{"time": "1681643999", "event": "Telegram Message Sent", "trigger_point(s)": "sqlite", "object":
     {"REPLACE INTO messages_v2 VALUES(19037, 961166549,..., 1676821892, n8<J'QY<J9xcRHey, I
    think something is wrong with your phone. You sent me a suspicious message.,...)"}}
{"time": "1676928079", "event": "Whatsapp Message Sent", "trigger_point(s)": "sqlite", "object":
     {"INSERT INTO message(...,sender_jid_row_id,...receipt_server_timestamp,text_data,...)
    VALUES (...4,18446744073709552000,...,18446744073709552000,DHL: Your parcel is arriving,
    track here: https://flexisales.com/dhl1eep7j88cc5z3,...)"}}
{"time": "1678038113", "event": "Signal Message Sent", "trigger_point(s)": "sqlite", "object": {
    "INSERT INTO message(view_once,receipt_timestamp,..,body..,recipient_id) VALUES
    (0,18446744073709552000,...,DHL: Your parcel is arriving, track here: https://flexisales.com
    /dhl18446744073709552000eep7j88cc5z3v,...,4)"}}
```

Listing 6.1: JIT-MF log entry sample containing SQL statements, generated while using WhatsApp, Telegram and Signal Android apps using SQLite JIT-MF-based Driver. Metadata which does not contribute to the investigation was redacted but can be found in the repository[13].

The models in Table 6.10 were generated based on the features described. Two models were created for each combination of anomaly detection method and feature or feature

[13]`https://gitlab.com/bellj/vedrando/-/tree/main/forensic_artefacts_collected`

set (in the case of K-means, PCA and Autoencoder). In the case of Google BigQuery's ARIMA_PLUS, log entries are automatically grouped in sixty-second time windows. For ARIMA_PLUS two different feature normalisation properties (Standard Scaler and Min Max Scaler[14]) are used, whereas for K-means, PCA and Autoencoder events are aggregated and counted every thirty and sixty seconds. Each model in Table 6.10 was created for every targeted app in the case study.

Table 6.10: Models generated based on the selected features.

Model	AD Method	Feature	Feature Options
M1	ARIMA_PLUS	Feature 1	Standard Scaler
M2	ARIMA_PLUS	Feature 1	Min Max Scaler
M3	K-Means	Feature 2 -11	Grouped by 30s
M4	PCA	Feature 2 -11	Grouped by 30s
M5	Autoencoder	Feature 2 -11	Grouped by 30s
M6	K-Means	Feature 2 -11	Grouped by 60s
M7	PCA	Feature 2 -11	Grouped by 60s
M8	Autoencoder	Feature 2 -11	Grouped by 60s

Anomalies are detected depending on the model used and the threshold set. ARIMA_ PLUS is a univariate time-series model that uses a single feature to detect anomalous data points across historical data. In contrast, K-means, PCA and Autoencoder models use multiple features for clustering (K-means) and dimensionality reduction (PCA, Autoencoder) that identify anomalies based on outliers and reconstruction loss. Each model supports a custom threshold for anomaly detection in Google BigQuery ML.[15] For ARIMA_PLUS models anomalies are identified based on the confidence interval for that timestamp. If the probability that the data point at that timestamp occurs outside of the prediction interval exceeds a given probability threshold, the data point is identified as an anomaly. Furthermore, since Google BigQuery returns the feature value, the detection algorithm implementation also checks that for the given anomaly found, *Feature 1* (the discrepancy between logs) is greater than 0. For the other models, anomalies are identified based on the value of each input data point's normalized distance to its nearest cluster. The data point is identified as an anomaly if that distance exceeds a threshold determined by the given contamination value. The contamination value defines the proportion of anomalies in the training dataset. This value ranges from 0.1 to 0.5, where 0.1 and 0.5 mean that 10% and 50% of the training data used to create the

[14]`https://cloud.google.com/bigquery/docs/reference/standard-sql/bigqueryml-preprocessing-functions`

[15]`https://cloud.google.com/blog/products/data-analytics/bigquery-ml-unsupervised-anomaly-detection`

input model, respectively, is anomalous. Whereas for the ARIMA_PLUS models, a *lower* threshold value makes data points more likely to be considered anomalous, for the other models, a *larger* contamination value (threshold) makes data points more likely to be considered anomalous.

High-level event reconstruction and correlation. Before executing the *Attack Detector* component of VEDRANDO, preliminary manual analysis was carried out to identify how low-level JIT-MF log entries can be combined to form more indicative high-level events. The analysis revealed that, in the case of SQLite API-based JIT-MF Drivers and the apps used in the case studies, a regex pattern for *JIT-MF_Logs* generated by each app could identify a log entry that reflects the actions of several JIT-MF low-level events generated after an action has occurred. Listing 6.2 shows the log entries obtained for the WhatsApp case study. In this case, the `SELECT`, `UPDATE`, and `INSERT OR IGNORE` log entries shown between *lines 2-21* all occur at the same timestamp and reflect a message send event. These SQL statements are executed to update several tables in the database that are affected by this event. For instance, log entries shown on *lines 13-16* set the last updated chat, which is reflected in the UI, whereas log entries on *lines 2-4* reflect execution related to opening a chat and drafting a message before sending. In this case, however, the high-level event that reflects the message sent with all its metadata (including message content) can be seen on *line 5*. This pattern was seen to be the case for WhatsApp message-sending activities. Therefore the regex used to identify a high-level WhatsApp messaging event was: `INSERT INTO message.*VALUES (.*)`, with resulting events also grouped by timestamp, under the assumption that a normal user cannot send more than one message at precisely the same millisecond. The result is the high-level event shown on *line 25*. The same approach was taken for the rest of the case studies.

```
 1 --- Low-level Events
 2 1676928079,SELECT _id, jid, serial, issuer, expires, verified_name,... FROM wa_vnames WHERE jid
       =...@s.whatsapp.net
 3 1676928079,INSERT INTO message_details(author_device_jid,message_row_id) VALUES (...)
 4 1676928079,INSERT INTO message_ftsv2(fts_jid,fts_namespace,content,docid) VALUES (...)
 5 1676928079,INSERT INTO message_ftsv2(fts_jid,fts_namespace,content,docid) VALUES (1 o,flo,dhl :
       your parcel is arriving , track here: https://flexisales.com/dhl29eep7j88cc5z3,?)
 6 1676928079,UPDATE chat SET last_read_receipt_sent_message_row_id=...,show_group_description=...
 7 1676928079,UPDATE chat SET last_read_receipt_sent_message_row_id=10,show_group_description=0,...
 8 1676928079,SELECT jid_row_id, type, message_count FROM frequent)q
 9 1676928079,SELECT user, server, agent, device, type, raw_string FROM jid WHERE _id=...
10 1676928079,SELECT user, server, agent, device, type, raw_string FROM jid WHERE _id=4s
11 1676928079,SELECT user, server, agent, device, type, raw_string FROM jid WHERE _id=null
12 1676928079,SELECT user, server, agent, device, type, raw_string FROM jid WHERE _id=15s
13 1676928079,UPDATE frequents SET message_count=... WHERE jid=... AND type=...
14 1676928079,UPDATE frequents SET message_count=2 WHERE jid=35679247196@s.whatsapp.net AND type=0
15 1676928079,UPDATE frequent SET message_count=undefined WHERE jid_row_id=... AND type=...
16 1676928079,UPDATE frequent SET message_count=2 WHERE jid_row_id=14 AND type=0
17 1676928079,COMMIT;
18 1676928079,BEGIN EXCLUSIVE;p
```

```
1676928079,INSERT OR IGNORE INTO message_link(message_row_id,link_id,chat_row_id) VALUES (...)
1676928079,INSERT OR IGNORE INTO message_link(message_row_id,link_id,chat_row_id) VALUES (...)
1676928079,COMMIT;

--- High-level Event
1676928079,INSERT INTO message_ftsv2(fts_jid,fts_namespace,content,docid) VALUES (1 o,flo,dhl :
     your parcel is arriving , track here: https://flexisales.com/dhl29eep7j88cc5z3,?)
```

Listing 6.2: Low-level event to high-level event mapping for the WhatsApp case study.

This manual process was also required to select the correlation regex keyword specific to each app. In these case studies, keyword regexes aim to extract the message content and identifier (ID). Therefore regex string patterns were defined for these two keywords for each app so that any message content or message ID found in log entries can be correlated with related events. Once the list of events correlated by message ID and text is produced, time-based correlation occurs (as shown in Algorithm 2) by retrieving events from other log sources within the period of the first and last correlated events. Listing 6.3 shows this for the WhatsApp case study. In the case of WhatsApp, the message content was retrieved via a position index within log entries that follow the regex: `INSERT INTO message.*VALUES (.*)`, where the content is found as position 15 within the parametrized values (the final group of the regex `(.*)`). The message ID, similarly, is found at position nine having regex: `^.*,([1-9]),`. Given the initial and last timestamps (`1676927672 - 1676928330`) retrieved for the list of logs correlated by message ID and content, a period is established within which a package install (*line* 2), comprising malware installation, is also disclosed. The same approach was followed in the rest of the case studies.

```
1 Timestamp, Log Entry, Forensic Source
2 1676927672,mid: 19039| uid: 961166549| in/out: out| message:
     DHL: Your parcel is arriving, track here:  https://flexisales.com/dhl?eep7j88cc5z3, Telegram
      Logs
3 1676927682,com.example.demo|2023-02-20 21:14:42+00|2023-02-20 21:14:42+00, Android Package Info
4 1676928312,INSERT INTO message(broadcast,message_add_on_flags,key_id,origin,participant_hash,
     sender_jid_row_id,recipient_count,message_type,chat_row_id,...,text_data,from_me,status,
     timestamp,received_timestamp) VALUES (0,0,99F904041FAFC5864CF7A590AB0F13BD,0,null,0,0,0,
     58975,...,
     DHL: Your parcel is arriving, track here:  https://flexisales.com/dhl1eep7j88cc5z3)x037j88cc
      5z3,0,1676928307435,0,?), JIT-MF_Logs
5 1676928330,DELETE FROM message WHERE _id IN (SELECT _id   FROM deleted_messages_ids_view   WHERE
      chat_row_id = 58975    AND sort_id>=31    ORDER BY sort_id ASC   LIMIT 100), JIT-MF_Logs
```

Listing 6.3: List of events correlated by message ID, content and time for the WhatsApp case study.

6.2.4 Attack investigation: Results

For each targeted app, the ground truth attack steps of the simulated app hijack were recorded as shown in Table 6.11 (a subset of the events shown in Table 6.8). VEDRANDO's *Events Collector* is evaluated based on the logs produced compared to those produced by plain EDRs (without JIT-MF installed). VEDRANDO's *Attack Detector* is evaluated based on its ability to detect the app hijack malware entry point, given evidence in *JIT-MF_Logs*.

Table 6.11: Comparison of ground truth attack steps disclosed and detected by a typical EDR and by VEDRANDO (columns 3 and 4, 5 and 6), respectively.

Event	**Event Description**	**Collected by EDR**	**Collected by VEDRANDO**	**Detected by EDR**	**Detected by VEDRANDO**
❷	Malware entry point	✓	✓	×	✓
❸	Malicious *demo.apk* installed	✓	✓	×	✓
❹	Link propagated to contacts	×*	✓	×	✓
❺	Propagated messages deleted	×*	✓	×	✓

✓ refers to *disclosed* attack steps.
× refers to *undisclosed* attack steps.
∗ these attack steps were recovered during the Skype case study *only*.

Artefacts recovered by *JIT-MF_Logs*. Table 6.11 summarises which critical attack steps executed on all targeted messaging apps were found in logs typically collected by an EDR and collected by VEDRANDO's *Events Collector* component that include *JIT-MF_Logs*.

The results show that VEDRANDO's *Events Collector*, using app-level virtualisation enhanced with JIT-MF Drivers, collects *JIT-MF_Logs* comprising evidence from memory from all the apps in the case studies without requiring app repackaging. In nine out of the ten case studies carried out (except for the Skype case study), critical attack steps (❹ and ❺) were *only* collected when considering JIT-MF forensic log sources. This evidence was located given knowledge of the ground truth. However, investigators and analysts investigating an attack scenario require a detection methodology that points to these specific events to detect anomalous behaviour. Specifically, events ❷ and ❸ can only be considered anomalous after having been correlated to events ❹ and ❺, collected solely by JIT-MF (except for one case study).

Reconstruction of attack steps. Now that results show that only *JIT-MF_Logs* uncover these anomalies, an analysis of the thresholds and model parameter selection is carried out to identify the ideal parameters for the *Attack Detector* to uncover these anomalies. Tables 6.12 and 6.13 show the effectiveness of the detection and correlation algorithm in VEDRANDO's *Attack Detector* component at reconstructing stealthy attack steps executed during the stealthy IM hijack case studies, with the input parameters defined in Section 6.2.3.2. For each model and threshold input combination, the average recall, precision and F1-scores are calculated to measure the overall accuracy of the reconstructed set of events returned by the *Attack Detector* when compared to the ground truth set of events executed by the attack. The F1-score combines precision and recall values. Therefore, the higher the F1-Score, the more accurate the list of attack steps returned by VEDRANDO's *Attack Detector*.

Table 6.12: Table showing the average F1-scores for the reconstructed attack steps across all case studies, generated by the combined anomaly detection and correlation algorithm when using ARIMA_PLUS models, with varying threshold values. The threshold values, in this case, are *inversely-proportional* to the allowance for anomaly probability.

Model	**Model Description**	**Threshold (anomaly probability)**			
		0.95	**0.90**	**0.85**	**0.80**
M1	ARIMA_PLUS using Standard Scaler	70.70%	73.08%	82.60%	82.60%
M2	ARIMA_PLUS using Min Max Scaler	67.96%	72.50%	82.17%	82.17%

Table 6.13: Table showing the average F1-scores for the reconstructed attack steps across all case studies generated by the combined anomaly detection and correlation algorithm when using K-means, PCA and Autoencoder models, with varying threshold values. The threshold values, in this case, are *proportional* to the allowance for anomaly probability.

Model	**Model Description**	**Threshold (contamination)**				
		0.1	**0.2**	**0.3**	**0.4**	**0.5**
M3	K-means Grouped at 30s	34.74%	51.19%	65.34%	68.09%	72.16%
M4	PCA Grouped at 30s	14.60%	57.55%	65.94%	77.61%	80.69%
M5	Autoencoder Grouped at 30s	34.74%	51.19%	65.34%	68.42%	72.16%
M6	K-means Grouped at 60s	32.60%	50.85%	70.68%	69.95%	76.90%
M7	PCA Grouped at 60s	17.35%	60.23%	78.99%	77.55%	83.53%
M8	Autoencoder Grouped at 60s	32.60%	50.85%	70.68%	70.09%	76.90%

The tables above show the averaged results over all the attack case studies carried out during experimentation. The results demonstrate that, overall, threshold parameter values with greater allowance for anomalies (< 0.9 for ARIMA_ PLUS and >0.3 for K-means, PCA and Autoencoder models) return more accurate reconstruction of attack steps. Specifically, three models (PCA models - M4 and M7 - and the ARIMA_PLUS model using a standard scaler - M1) result in an F1-score of 80% and have *100% recall*

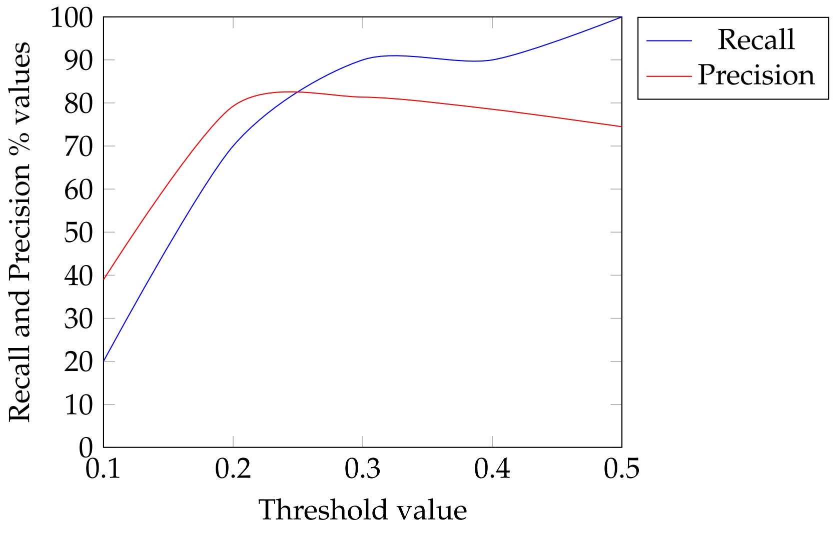


Figure 6.13: Average for recall and precision values for a given threshold, when using PCA model M7 across the ten case studies.

value; that is, full attack step reconstruction.

Further analysis of the results obtained by these three models revealed that the average recall value across apps was increasing at a faster rate than the precision value was decreasing. This is because, for individual case studies (which vary depending on the model used), a more lenient threshold value was required to obtain the same recall value that other apps obtain at less lenient threshold values. Figure 6.13 shows this for the specific case of the model input parameter resulting in the best overall F1-score value (M7). In this case, 90% of the apps used in the case studies reached an average of 100% recall value on the reconstructed set of attack steps when the threshold was set at 0.3. However, WhatsApp Business attack steps were only detected as anomalies when the threshold was set to 0.5.

Overall, results show that PCA works well with the features selected to detect anomalies in *JIT-MF_Logs*, due to the high F1-scores when using both PCA models. Crucially, by using PCA (an existing anomaly detection algorithm available to SOC analysts) with a 0.5 threshold, using the set of features described in Table 6.9 and correlation settings defined in Section 6.2.3.2, the detection algorithm used by VEDRANDO can fully reconstruct attack steps of app hijack attacks with relatively high precision (~72.4%)across all case studies based on evidence collected from *JIT-MF_Logs*.

The sensitivity of the implemented detection and correlation algorithm to the threshold value given as a parameter was evaluated using Wilcoxon signed rank test.[16] Results

[16]`https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test`

Table 6.14: *p*-value results for Wilcoxon signed rank test given pairwise threshold values for K-means, PCA, Autoencoder and ARIMA models.

K-means, PCA, Autoencoder models			**ARIMA models**		
Pairwise thresholds	*p*-value	Statistically significant? (*p*-value < 0.05)	Pairwise thresholds	*p*-value	Statistically significant? (*p*-value < 0.05)
0.1, 0.2	0.03501	Yes	0.8, 0.85	1	No
0.2, 0.3	0.03501	Yes	0.85, 0.9	0.3333	No
0.3, 0.4	0.4375	No	0.9, 0.95	0.3333	No
0.4, 0.5	0.03125	Yes			

shown in Table 6.14 demonstrate that the change-in-value of F1-scores between threshold values for K-Means, PCA and Autoencoder is overall statistically significant (with *p*-value less than 0.05 [51]). Therefore when using such models, the algorithm is considered sensitive to the threshold set. On the flip side, the change in threshold values for ARIMA_PLUS models values is not. This is already visible in Table 6.12, where the F1-scores for the most lenient threshold values (0.8,0.85), remain the same. Given this, one still has to consider that the size for the set of ARIMA models, comprised two models, whereas that for K-Means, PCA and Autoencoder models was larger (six models in total), offering a more stable result.

6.3 Summary of findings

This chapter demonstrated that the resulting *JIT-MF_Logs* produced by minimally invasive JIT-MF Driver implementations enable full attack step recovery, both when using traditional mobile forensic tools and EDR tools. Thus, the case studies show evidence supporting the hypothesis that minimally invasive attack step recovery from volatile memory can produce accurate forensic timelines that aid forensic investigators during app hijack incidents. This was demonstrated in qualitative case studies comprising benign IM hijack scenarios inspired by realistic malware. Generic API-based JIT-MF Driver implementations were used, which, while being the least app-invasive, produce accurate *JIT-MF_Logs* (as demonstrated by results obtained from exploration described in previous chapters). This API-based JIT-MF Driver specification enabled experimentation to be carried out on a range of IM apps while using stock devices and apps.

Table 6.15 summarises the experiments carried out during this evaluation. The table outlines each experiment (denoted by the section in which it was described) and its main objectives in relation to the hypothesis presented in this thesis (that is, that app-specific artefacts from memory enable the complete recovery of attack steps in the case of app hijack attacks). The tools used in each experiment are listed along with the main conclusions and impact of the results.

The overall results show that *JIT-MF_Logs* improve on state-of-the-practice forensic tools and EDRs in the context of attacks following the app hijack threat model by collecting evidence of attack steps in memory that is not recorded by standard forensic tools (Belkasoft and XRY). Results from experimentation demonstrate that only *JIT-MF_Logs* contained evidence of message propagation, including object metadata (Section 6.1). Having obtained results showing that evidence of app hijack attack steps can only be found in *JIT-MF_Logs*, the next experiment described an EDR and detection workflow that is sensitive to evidence in *JIT-MF_Logs* (Section 6.2). The results from the experiment showed that evidence in *JIT-MF_Logs* can be detectable as anomalous, using anomaly machine learning with a set of specific parameters, and a correlation and detection algorithm can be used to fully recover attack steps in the case of a messaging hijack app.

Table 6.15: Summary of experiments carried out to evaluate *JIT-MF_Logs* with state-of-the-art forensic tools, within the context of IM app hijack attacks.

Experiment	**Objective**	**State-of-the-art Forensic tools**	**Main conclusions**	**Impact**
Section 6.1	Demonstrate the ability of a JIT-MF tool to contribute to forensic timelines generated by state-of-the-practice mobile forensic tools	Belkasoft, MSAB XRY	• *JIT-MF_Logs* uniquely contribute to forensic timelines generated by state-of-the-practice mobile forensic tools through recorded attack steps that are deleted or inaccessible to other forensic tools. • Both MSAB XRY and Belkasoft proprietary tools could not recover critical metadata related to hijacked attack steps, unlike MobFor. This JIT-MF tool was able to produce the missing metadata.	• Artefacts found in *JIT-MF_Logs* complement those retrieved by forensic tools. • Experimentation confirmed that app repackaging to include JIT-MF Drivers for forensic enhancement can conflict with app protection measures related to anti-repackaging checks.
Section 6.2	Demonstrate how full attack step recovery of benign hijack attacks is only possible using *JIT-MF_Logs*.	ReLF EDR using GRR Server	• Attack steps executed through app hijacking and recorded in *JIT-MF_Logs* are detectable as anomalies. • *JIT-MF_Logs* carry sufficient features to be correlated with the attack's entry point on the device.	• Full attack step recovery (up to malware entry point) is possible for app hijack attacks, using *JIT-MF_Logs* alongside other forensic sources typically collected, allowing for attack step remediation. • Non-invasive approach avoiding app unpackaging and repackaging is possible through API-based JIT-MF Driver and JIT-MF installation at app level leveraging app-level virtualisation.

7 Discussion

The research question and hypothesis set in this thesis called for an exploration to determined the minimum level of invasiveness possible for the timely collection of app-specific artefacts from memory without compromising forensic timeline accuracy. It also involved evaluating artefacts in *JIT-MF_Logs* alongside those collected by state-of-the-art forensic tools. To this end, experimentation comprised implementing the JIT-MF framework (described as a means to explore the research question of this thesis) and analysing the results from case studies comprising simulated messaging hijack attacks. This involved the creation of JIT-MF Drivers with *Evidence_object*s and *Trigger_point*s from different layers of the stack, JIT-MF tools and JIT-MF-enhanced EDRs.

This chapter presents an overall analysis of the experimentation results and what these results demonstrated in terms of the initial objectives set for experimentation (Section 7.1) in line with the research question, along with the threats to their validity and experimentation limitations (Section 7.2). Finally, an overview of the development effort and challenges involved in developing the JIT-MF tools used during experimentation is also presented (Section 7.3).

7.1 Analysis of experimentation results

Timely-collected evidence from volatile memory is necessary for the complete attack step recovery of app hijack investigation. The experiments involved simulated app hijack scenarios that emulate realistic malware [7]. Results from case studies used during exploration showed that typically collected forensic sources lacked the necessary app-specific artefacts to disclose attack steps comprising hijacked app functionality. The presence of app-specific artefacts does not directly imply a complete forensic timeline comprising all the recovered attack steps, as this relies on several factors, including the investigator's analytical abilities and the forensic analysis tools used. However, if these artefacts are missing or uncollected in the first place, then complete forensic timeline generation is guaranteed not to be a possibility. Results from the case studies used during experimentation show that in the case of app hijack attacks, app-specific artefacts contribute to a complete forensic timeline and therefore are necessary for attack step reconstruction and remediation. However, the messaging application logs of many apps used during experimentation did not disclose this evidence. *JIT-MF_Logs* were used instead to compensate with evidence collected from memory. Application developers could aid in this respect by including a heightened level of forensic readiness to complement the current security trend by design. Such capabilities enable forensic investigations, yet it is difficult to predict which events should be logged or can be hijacked. Simultaneously, logging all possible events related to application functionality of interest would render any app unusable. JIT-MF can alleviate this challenge whereby it can be possible for application developers to embed JIT-MF capabilities within apps, having configurable options that enable investigators or even developers to modify the *Evidence_object* and *Trigger_point* depending on the functionality of interest that needs to be logged in the case of a specific app hijack attack scenario. JIT-MF can be used in a setting similar to Runtime Application Self-Protection (RASP) [60] solutions used post-deployment for app hardening to monitor possibly unauthorised access and interactions; however, for timely dumping of evidence in memory, instead.

Evidence from memory alone is not enough for full attack step recovery. Results from the experiments show that while the evidence in *JIT-MF_Logs* is necessary, it is not sufficient in isolation for a complete reconstruction of attack steps. This has been demonstrated in other research [89], where advanced cyber attacks execute their attack steps in multiple stages, aiming to be stealthy and elusive. This means that any evidence of attack steps is dispersed across multiple sources and requires a process of log event correlation for full attack step reconstruction. Case studies with other forensic tools, EDRs, and anomaly detection mechanisms show that *JIT-MF_Logs* contain missing evidence of attack steps not found in other sources. Moreover, the evidence in *JIT-MF_Logs* carries enough features in metadata that this evidence can be correlated to other forensic sources collected to generate a complete forensic timeline comprising all attack steps, including the malware entry point. This implies that incident response is then able to remediate all attack steps.

App-specific evidence of app hijack attacks can be timely collected from memory by identifying app objects that need to be dumped and associated app instructions that result in these objects being in memory. Experiment results have shown that this is possible through function hooking of app instructions to dump the identified object in memory. This was implemented through *Trigger_point*s and *Evidence_object*s. Yet, the process of function hooking so far involved device or app-invasive approaches. Timely collecting of app-specific artefacts from memory can be non-invasive by instrumenting the app runtime level and identifying publicly-documented APIs that handle application functionality of interest. In all the experiments, JIT-MF was implemented at the app level avoiding device rooting and even shown to operate using app-level virtualisation to avoid app repackaging while still timely collecting evidence from memory. This means that stock apps and devices can be used, ensuring that the approach is compatible across different devices and apps used by potential victims and that the device and app's default security are not weakened. App invasiveness also concerns JIT-MF Driver development regarding *Trigger_point*s and *Evidence_object*s selection. App-specific artefacts required as evidence for app hijack attacks call for app-specific *Evidence_object*s which can be linked to app-specific instructions (*Trigger_point*s). Yet identifying such objects and instructions requires compiled code analysis to gather knowledge of app internals comprising app-invasive approaches such as app unpacking. Such approaches not only break stock apps but also means that the JIT-MF Driver is app-specific, making for an infeasible approach from an operational point of view.

The key results from the exploration of JIT-MF positioning demonstrate that JIT-MF Driver development can also be non-invasive without compromising forensic timeline accuracy in the case of messaging hijack attacks. These drivers can be generalised by selecting *Trigger_point*s and *Evidence_object*s from the API layer of the Android technology stack, which typically provides publicly-available documentation. This means that triggering app-specific artefacts from process memory does not necessarily require an app-specific approach that is infeasible to carry out across multiple apps. While the resulting logs produced require some parsing due to the application-specific usage of the API, the execution of experiments has shown that the effort required is reasonable since the API used is publicly documented. During experimentation, it was noted that the developed API-based JIT-MF Drivers rendered the app more stable regarding performance. Yet further exploration is required to assess whether this depends on the API selected or a reflection on the selection of the *Trigger_point*s and *Evidence_object*s from the Android API stack layer.

7.2 Threats to validity

Due to the rigorous setup required to emulate the app hijack attack scenarios, which are app-specific by design, and the forensic investigation environment, the experiments were qualitative; i.e., the case studies were limited in scope (messaging app category) and number. Furthermore, the analysis and JIT-MF Driver development involved in the experiments relies on the investigator's analytical abilities, which is impossible to isolate, and which was crucial to recover *Evidence_object*s in memory that could be related to hijacked app functionality.

While the concept of the JIT-MF framework is applicable across app hijack attack scenarios targeting different app categories, further experimentation involving other app categories would be required to assess the implication on *Trigger_point* and *Evidence_object* selection when enhancing these apps. Within the case studies involved in experimentation, selecting *Trigger_point*s and *Evidence_object*s followed from app storage and network interactions. Yet it is possible that instructions and objects related to UI transformation for other app categories would provide more insight into attack steps. While APIs for UI functionality exist, UI-related instructions would likely be invoked more frequently due to the number of UI elements involved in an app. This leads to increased *Trigger_point* frequency and possibly impacts the app stability more. In this case, the sampling strategies described in Section 5.5 can be adapted to the scenario. It is also possible that while still leveraging APIs for JIT-MF Driver development, generalising the same driver across apps is more challenging when focusing on UI APIs. For instance, the scope of the SQLite API is relatively narrow in that apps use it to store essential data typically linked to app functionality of interest. UI functionality, on the other hand, is broad and specific to the app.

With regards to the number of messaging apps involved in the experimentation, while the apps used (specifically instant messaging apps) comprise a large majority of the messaging market share,[1] the number of apps still was relatively small and comprised a single category; that is messaging.

Despite simulating the environment within which JIT-MF is expected to operate as best as possible, the results reflect experiments conducted in a lab setup. This is especially true concerning normal traffic generated through individual app usage patterns. This should not impact the accuracy of the app-specific artefacts collected by JIT-MF Drivers. However, app stability may be affected, depending on app usage. Anomaly detection features and threshold parameters must also be adjusted to account for normal app usage patterns.

Internal threats to validity comprise using emulated devices for experimentation and static analysis for determining common API usage across apps. Initial experiments leveraged emulated devices for ease of automation. Although emulated environments enable device rooting through a simple command, this was not leveraged to enable the operation of JIT-MF. This is mainly reflected in experiments Sections 5.4, 6.1 and 6.2, where stock unrooted Android devices were used for experimentation. On the other hand,

[1]`https://www.similarweb.com/blog/research/market-research/worldwide-messaging-apps/`

the performance of emulators is expected to be worse than actual devices. Therefore where storage and performance results indicate possible degradation, the results may improve on a physical device.

The use of SQLite API enabled the generalisation of a non-invasive JIT-MF Driver. SQLite is a popular library many developers use, yet AppBrain was used to identify popular APIs across a category of apps for the experiments described. AppBrain is closed-source; however, it provides insight into how it derives information regarding Android apps on Google PlayStore. It analyses all Android apps on Google Play and performs some analysis on the package file of the app (APK). Statistics regarding development tools and libraries usage are obtained by matching package names inside the apps “with known package names from development tools. Therefore, these statistics reflect whether the code of a certain library is present in an app." [14]. While this does not guarantee that a particular app actively uses a library, “it still gives a good idea of the market share and the list of the top Android development tools." [14]. Furthermore, regarding the permanence of API usage across app versions, a static signature was used following the logic of how apps interact with the underlying infrastructure. Being a static check, this does not guarantee that the app actively uses the code, as there is also the possibility that the code or libraries found reflect dead or legacy code. That said, the API identified as generic across messaging apps (SQLite) was sufficient to develop a generic JIT-MF Driver that functioned across ten of the most popular IM apps.

Beyond the setting of the Android platform, the concept of JIT-MF can be applied in the context of other operating systems, yet both the challenges and the concept itself change in the context of other platforms. Similarly to how the app hijack threat model hijacks app functionality to offload attack steps and evade detection, living-off-the-land techniques have been shown to take advantage of functionality in binaries and libraries native to the OS to carry out malicious steps. While different attack vectors may be used, the threat model remains the same: leveraging existing device or workstation functionality to offload attack steps that aid malware in evading detection. A case in point is the malicious use of PowerShell, a powerful native Windows tool that can execute commands as part of attack steps [20]. Similarly to the app hijack threat model, these steps evade traditional defences since this tool is legitimate to execute Windows commands. Therefore, the threat and need for attack step reconstruction is still present on other platforms. The availability and access to third-party application logs may be easier on other workstations, as access to them is not bound by the OS security model, as is the case with Android. However, issues related to whether or not the contents of these logs are sufficient for forensic investigators and subject to malware tampering are not specific to a platform but related to applications in general. Therefore, timely collected evidence from memory can still contribute to such attack scenarios. The challenges of EDR deployment differ significantly on workstations from an Android OS since the security model of workstations does not rely on limiting administrator privileges. As such, challenges on Android related to invasive approaches for evidence collection are not as much of a concern on general-purpose workstations. However, EDRs still rely on known attack vectors for detection and monitoring. Therefore, unless benign app instructions are also monitored for potential hijack activity, evidence in memory of hijacked attack steps would still go uncollected.

7.3 JIT-MF Drivers, tool and EDR development

To assess the JIT-MF framework, experimentation involved implementing various JIT-MF Drivers, a JIT-MF-based tool (MobFor) and an enhanced EDR, complete with detection and correlation workflow (VEDRANDO).

The development of JIT-MF Drivers for use in experiments was narrowly scoped. The selection of *Trigger_point* and *Evidence_object* focused on different stack layers and functionality related to network or storage interactions. Yet, this still comprised a large number of possible instructions. JIT-MF Driver development process first required familiarisation with the app to identify which app functionality of interest could be hijacked. *Trigger_point* and *Evidence_object* selection comprised identifying the *Evidence_object* first, then identifying a set of instructions that could be potential *Trigger_point*s. Preliminary runs informed the decision regarding which *Trigger_point* is ideal, in terms of minimum frequency yet high accuracy, to dump the *Evidence_object* timely. Once *JIT-MF_Logs* are collected, a post-processing process involved ensuring no duplicate entries and parsing of *Evidence_object*. Parsing was minimal in the case of app-specific *Evidence_object*. However, API-based drivers generated *JIT-MF_Logs* that required parsing since the objects reflected app objects as API-specific parameters. While apps make app-specific use of APIs, the parsing effort was relatively straightforward due to publicly available API documentation and the well-known schema of these apps.

MobFor is a JIT-MF-based tool[2] built for experimentation, whose development considered feedback from cyber security blue teams that are practising incident response teams. This tool enhanced apps and collected *JIT-MF_Logs* to generate an accurate forensic timeline, along with evidence collected by other mobile forensics tools. As per feedback, it enables the concept workflows that handle the entire process from enhancing apps to collecting *JIT-MF_Logs* and transferring evidence to a given server for analysis. MobFor

[2]https://gitlab.com/mobfor/mobfor-project

also enabled a chain of custody, comprising a log file with the list of commands executed on the device and hashes of outputs at each workflow stage. While MobFor does not cater for forensic analysis as the tool was developed to collect evidence from a connected device having an enhanced app. Yet the outputs from the device (*JIT-MF_Logs*) are parsed and sanitised from any duplications to produce a JSON file that can be uploaded into Timesketch to produce a forensic timeline of *JIT-MF_log* events.

App repackaging is mitigated to enable the use of stock apps through app-level virtualisation. Current app-level virtualisation frameworks are still in their infancy. This means they are limited in functionality, rendering them non-functional or limited when using specific apps that require Google Play services, for instance. Further app-level virtualisation limitations exist when the targeted apps are system apps, which this work has not addressed. Therefore further development effort is needed to create more stable, open-source virtualisation frameworks before the solution proposed in this thesis can be fully realised.

Similarly to popular memory forensics tools like Volatility [132], JIT-MF adopted the notion of plugins in JIT-MF Drivers specific to the app and attack scenario pairs. This enables the JIT-MF framework to be applied in different hijack attack scenarios. Like Volatility plugins, the JIT-MF Drivers allow for collecting app-specific parsed data, giving forensic investigators additional context. The use of underlying technology enablers (such as Frida in the prototype implementation of JIT-MF) allows for a platform-aware implementation required by memory forensics tools. Existing research focusing on trigger-based memory collection is similar to our work addressing the ephemerality of evidence in memory through trigger-based dumps. Yet these works are not concerned with device [122] or app [130] invasiveness, thus foregoing device security or requiring an operationally infeasible approach across apps, especially those that employ anti-repackaging checks. JIT-MF tools mitigate these challenges by leveraging the app layer and app-level virtualisation for implementation. By operating at this layer and using *Evidence_object*s from the API layer, the effort of app or tool comprehension and carving app or tool-specific evidence from memory is lessened.

Regarding challenges related to memory forensics, JIT-MF tools address the timely collection of ephemeral objects and mitigate app and device invasiveness required by standard memory forensics tools on Android. Similarly to how Volatility plugins for identifying application-specific analysis require effort to develop, selecting *Evidence_objects* and *Trigger_points* for the right attack scenario still requires insight and forensic analytical skill. Moreover, JIT-MF tools face further challenges due to their dependency on technology enablers, namely its reliance on the underlying hooking framework that enables the JIT-MF Driver runtime, which may be subject to changes to the Android runtime. The same can be said for the app-level virtualisation technology enabler that JIT-MF tools rely on to avoid app repackaging.

Throughout experimentation, the use of rich technology enablers enabled the experimentation to focus on the selection of *Evidence_objects* and *Trigger_points* and whether or not these could be categorised to enable JIT-MF Driver development while reducing invasiveness and maintaining accuracy. The selection of an API JIT-MF Driver and the use of app-level virtualisation frameworks allowed for a broader experiment encompassing multiple messaging apps and avoided app-specific efforts to mitigate any anti-repackaging that proprietary apps employ as part of their default security protection. However, most case studies involved apps that ran using a single process. Using apps that run multiple processes in future experiments could disclose potentially interesting insights about limitations or further development required for JIT-MF to work with such apps.

8 Conclusions

This thesis set out to address the limitations of existing incident response tools and invasive memory forensics approaches available to investigators when responding to the app hijack threat model on Android. Attacks following this threat model were shown to have the potential for stealth as they hijack benign app functionality to carry out attack steps, thus evading detection. Furthermore, these attacks can potentially delete any forensic footprints from the device and are typically used during incident response. Existing incident response tools that perform memory forensics do not address the ephemerality of evidence in memory and require invasive approaches (such as device rooting), typically used on an attacker's device rather than a victim's device, which weakens the device's overall security.

The exploration carried out in this thesis aimed to answer the research question: **How can attack steps offloaded to hijacked app functionality be timely recovered from volatile memory of stock Android devices, in the least invasive way possible concerning both the device and the hijacked app?** The hypothesis proposed that timely and non-invasive attack step recovery from volatile memory is possible by implementing memory introspection through dynamic binary instrumentation at the app level and focusing on lower layers of the tech stack to identify triggers and objects that could represent hijacked app functionality.

This chapter concludes this thesis by summarising how existing challenges and limitations related to Android memory forensics and incident response were overcome through a novel concept for responding to app hijack attacks on Android using triggered memory dumps (Section 8.1). As a result of the experimentation carried out, further research gaps are identified and are presented (Section 8.2) along with the final thoughts and main conclusions of this work (Section 8.3).

8.1 Timely, non-invasive evidence collection from Android volatile memory

App hijack attacks have been identified as a stealthy and evasive threat, which comprises malware that offloads attack steps to legitimate benign app functionality. This study has shown that on-device detection techniques cannot detect such malware, thus addressing the first object of this thesis ***O1***. Moreover, the forensic sources collected during the incident response process, which follows much later after an alert is raised (once malicious behaviour is detected through its consequences), are incomplete. Existing tools and forensic techniques rely on stored evidence collected from app data and devices, which typically require device rooting to collect evidence from internal storage. Not only has this evidence been proven futile due to the reduced forensic footprints of such stealthy attacks, but the victim is left with a device whose security has been compromised during the investigation. Any evidence of the attack can only be found in memory. Yet, the stealthiness of such attacks means that any ephemeral evidence in volatile memory would no longer be present when the victim notes the malicious behaviour. Furthermore, memory forensics on stock Android devices remains a challenge that investigators must adapt to using invasive approaches such as custom kernels. An optimal solution can faithfully reconstruct the attack steps timeline, even if the stealthy malware has erased these and is short-lived in memory, while able to operate within the constraints of stock Android devices and apps.

This thesis presents a novel approach, Just-in-time Memory Forensics (JIT-MF), that uses triggered memory dumps to timely collect evidence objects from memory by enhancing targeted apps in the preparation stage of incident response before an incident occurs. Challenges related to device rooting and collection of app-specific evidence as necessary by the app hijack threat model are mitigated by taking an app-level approach. App runtime is modified to include instrumentation that triggers memory dumps containing evidence of hijacked app steps. Yet the concept presented new challenges related to the selection of evidence objects, which indicate the presence of a potential attack step, and triggers, the instructions invoking the existence of these objects in memory. The challenges arose from the app-specific code comprehension potentially required to select them. The creation of this novel approach was the first step towards addressing the second objective ***O2*** of this thesis, which allowed an exploration of the technology stack for a minimal app and device-invasive approach.

An exploration involving different layers of the Android technology stack was carried out to determine which layers are ideal for sufficient app-specific detail yet do not impose an app-invasive approach, contributing to the second objective of this thesis ***O2***. The exploration took into account different layers in the stack, the resulting accuracy of the objects dumped and the app's stability. The exploration demonstrated that this novel non-invasive timely approach to app-specific evidence collection from volatile memory is possible and operationally feasible across various apps without compromising accuracy, by leveraging the API layer of the technology stack for *Evidence_object* and *Trigger_point* selection. Documentation of API layer code is typically publicly-available, rather than requiring compiled code analysis, and is leveraged by multiple apps for app functionality of interest. Experimentation was also carried out to verify that app stability can be achieved through sampling while maintaining evidence accuracy.

Crucially, an evaluation with state-of-the-art forensic tools (addressing the third objective of this thesis ***O3***) showed that this novel approach enables the completion of forensic timelines, using timely evidence from memory while working with stock devices and stock apps through runtime manipulation. Furthermore, through the development of a novel correlation and detection algorithm for *JIT-MF_Logs*, the evidence of hijacked attack steps found in these logs are detectable as anomalies and carry sufficient features to be correlated with events leading up to the attack's entry point on the device.

8.2 Future work

The exploration of JIT-MF provided initial insight into this approach, showing that triggered memory dumps taken at the app level are promising for the timely collection of evidence related to app hijack attacks while also being non-invasive. However, a larger-scale exploration is needed to evaluate this concept in a real-world setting. This calls for realising further attack scenarios that follow the app hijack threat model, specifically targeting categories of apps other than messaging. An evaluation of JIT-MF in these scenarios could shed light on how the conclusions concerning the operational aspect of JIT-MF that were determined in this thesis translate to app hijack attacks targeting other app categories. For example, exploring whether or not the generalisation of JIT-MF Drivers through selecting trigger instructions from the API layer applies to other app categories and the effect of enhancing different app categories on device resources. An analysis could be made to conclude whether specific stack layers benefit a specific app category and whether or not this applies to most apps within that category. Furthermore, the relationship between the evidence object and the attack step may vary from one category to another, and perhaps further conclusions could be drawn from the evidence objects. Therefore, further experimentation could look into developing app or app category profiles that indicate how the evidence object could be interpreted for a specific app and if this can be generalised to the app category.

Another aspect of a larger-scale evaluation comprises evaluating JIT-MF Drivers in a more realistic setting that exposes typical usage patterns of the targeted apps. This would be useful to explore the effects of JIT-MF on app stability and device resources for long-term use. The result could be used to identify log collection duration based on typical app usage patterns that could alleviate the strain on device resources and app stability due to accumulated logs.

Further analysis is required regarding the usability aspect of JIT-MF. App and device performance and stability contribute to this aspect. However, another concern is privacy. Evidence of app-specific artefacts collected from memory may contain sensitive data the user does not wish to expose. The premise of this novel approach is that it operates in a setting aiming to aid end-users that may eventually become subject to stealthy app hijack attacks. While it is within the interest of the device owner that the evidence is collected and analysed when and within the parameters of the incident, the collection of sensitive data may incur additional heavy lifting on the processing side to comply with GDPR standards. To this end, future work can look into the possibility of adopting privacy-aware forensics [45] through which the necessary evidence to reconstruct stealthy attack steps is collected from memory while sensitive information is withheld to protect users' privacy. Searchable Symmetric Encryption (SSE) [38] can be adopted to allow privacy-preserving forensic analysis of evidence by encrypting the contents of *JIT-MF_Logs* in such a way that it is concealed yet still searchable by investigators, through pre-generated indices. In the case of messaging hijack attacks, indices can be generated based on the keywords that identify texts to be suspicious, for instance, the text format of URL addresses known to be propagated by specific malware or suspicious phone numbers found in the list of contacts. Investigators can then flag suspicious events that a malicious actor may have caused without access to the device owner's private data.

From a practical point-of-view, other usability issues concern installing JIT-MF and shipping logs, especially within large organisations comprising multiple personnel equipped with individual devices. Mobile Device Management (MDM) tools provide a solution for the administrative management of devices in large organisations. Within the context of *JIT-MF_Logs*, such tools can facilitate the automatic shipping of logs followed by a possible proactive response, for example, through SIEM integration. Therefore, a possible research direction for this work could look into embedding the concept of JIT-MF within MDM tools for ease of deployment and management in large-scale settings.

Full attack step reconstruction was possible through a detection and correlation algorithm enabled by generic online services with machine learning capabilities that offer default machine learning. Further work in this area could focus on *JIT-MF_Logs* log analysis, which could look into feature selection and identify which machine learning models and parameters are appropriate for specific apps and attack scenarios. This could also aid in utilising *JIT-MF_Logs* more proactively. So far, this thesis explored the value of timely-collected evidence from memory for app hijack attacks in a reactive setting. That is, a victim grows suspicious of malicious behaviour on their device after some of the consequences of the malware are revealed. Subsequently, an alert initiates a forensic investigation as part of the incident response. Therefore another research direction could explore the possibility of automated shipping of *JIT-MF* logs and detection through JIT-MF-sensitive anomaly detection. Automation can be carried out on periodically-shipped logs to alert the user if the evidence in *JIT-MF_Logs* may indicate possible app hijack malware.

8.3 Final remarks

The work presented in this thesis aimed to address the limitations of incident response on Android within the context of the app hijack threat model. The novelty of this work lies in a new concept, Just-in-time Memory Forensics (JIT-MF), devised to tackle the ephemerality challenges present in volatile memory without imposing invasive approaches on the device or app. Unlike other mobile forensics and EDR tools, JIT-MF can timely collect app-specific evidence from memory through trigger points. By enhancing an app's runtime with instrumentation capabilities, JIT-MF can timely dump objects from memory as evidence of hijacked attack steps that may not be found stored in the app data or on the device, aiding investigators during incident response. Moreover, this evidence carries sufficient features to indicate which objects may be the result of hijacked app functionality. Crucially for the device owner, and unlike other state-of-the-art forensic tools, enhancing the device does not require invasive techniques comprising device rooting or app repackaging that weaken the overall device security and default app protections.

This work has shed light on the usefulness of timely collected evidence from memory, specifically demonstrated for case studies involving messaging hijack attacks. This was achieved through complete investigation setups involving tools developed based on this concept, simulated app hijack attacks and state-of-the-art forensic tools. The result of the case studies showed that for attack scenarios following the app hijack threat model, evidence from memory is crucial to detect and recover all attack steps. Whilst this work has also resulted in several possible research directions and identified further exploration required to adapt to real-world scenarios, the concept of Just-in-Time Memory Forensics already presents an opportunity to advance existing mobile incident response practices to produce more accurate forensic timelines in the case of app hijack attacks on Android.

References

[1] T. C. A. Hoog and T. Anderson. AFLogical OSE: Open source Android Forensics app and framework. https://github.com/nowsecure/android-forensics, 2015. Accessed: 18.02.2023.

[2] A. Aarness. What is EDR? Endpoint Detection & Response Defined. https://www.crowdstrike.com/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/. Accessed: 02.03.2023.

[3] M. Alecci, R. Cestaro, M. Conti, K. Kanishka, and E. Losiouk. Mascara: A Novel Attack Leveraging Android Virtualization. *CoRR*, abs/2010.10639, 2020.

[4] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. Richard III. DroidScraper: a tool for Android in-memory object recovery and reconstruction. In *RAID*, pages 547–559, 2019.

[5] A. I. Ali-Gombe, B. Saltaformaggio, D. Xu, G. G. Richard III, et al. Toward a more dependable hybrid analysis of Android malware using aspect-oriented programming. *computers & security*, 73:235–248, 2018.

[6] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *USENIX Security Symposium*, pages 3005–3022, 2021.

[7] Amer Owaida. Wormable Android malware spreads via WhatsApp messages. https://www.welivesecurity.com/2021/01/26/wormable-android-malware-spreads-whatsapp-messages, 2021. Accessed: 9.11.2021.

[8] D. Andriesse. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly.* no starch press, 2018.

[9] Android Developer's Guide. https://developer.android.com/guide/topics/data/autobackup, 2020. Accessed: 7.7.2023.

[10] Android Developer's Guide. https://developer.android.com/guide/topics/data/backup, 2023. Accessed: 7.7.2023.

[11] Application Sandbox. https://source.android.com/docs/security/app-sandbox, 2023. Accessed: 7.7.2023.

[12] C. Anglano, M. Canonico, and M. Guazzone. Forensic analysis of Telegram Messenger on Android smartphones. *Digital Investigation*, 23:31–49, dec 2017.

[13] T. Apostolopoulos, V. Katos, K.-K. R. Choo, and C. Patsakis. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems*, 116:393–405, 2021.

[14] AppBrain: Monetize, advertise and analyze Android apps. https://www.appbrain.com/stats/libraries, 2022. Accessed: 23.08.2022.

[15] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Breaking and fixing the Android launching flow. *Computers & Security*, 39:104–115, 2013.

[16] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. ARTist: The Android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.

[17] Belkasoft. How to acquire data from an Android device using APK downgrade method. `https://belkasoft.com/Android_APK_downgrade_method`. Accessed: 8.11.2021.

[18] Belkasoft. Belkasoft evidence centre x. `https://belkasoft.com/x`, 2021. Accessed: 8.11.2021.

[19] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. I. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III. Tipped off by your memory allocator: Device-wide user activity sequencing from Android memory images. In *NDSS*, 2018.

[20] C. Black. 'PowerShell' Deep Dive: A United Threat Research Report. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/docs/vmwcb-report-powershell-deep-dive.pdf`, 2016. Accessed: 25.07.2023.

[21] BlackRock - The Trojan that wanted to get them all. `https://www.threatfabric.com/blogs/blackrock_the_trojan_that_wanted_to_get_them_all.html`, 2020. Accessed: 25.05.2022.

[22] J. Bremer. x86 API hooking demystified. *Development & Security*, 2012.

[23] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 133–144, 2012.

[24] Campbell, Christopher and Graeber, Matthew. Living Off the Land: A Minimalist's Guide to Windows Post-Exploitation. `http://www.irongeek.com`, 2013. Accessed: 24.03.2021.

[25] M. Cao. *Understanding the characteristics of invasive malware from the Google Play Store*. PhD thesis, University of British Columbia, 2022.

[26] A. Case, R. D. Maggio, M. Firoz-Ul-Amin, M. M. Jalalzai, A. Ali-Gombe, M. Sun, and G. G. Richard III. Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics. *Computers & Security*, 96:101872, 2020.

[27] A. Case and G. G. Richard III. Memory forensics: The path forward. *Digital Investigation*, 20:23–33, 2017.

[28] Y. Chabot, A. Bertaux, C. Nicolle, and M.-T. Kechadi. A complete formalized knowledge representation model for advanced digital forensics timeline analysis. *Digital Investigation*, 11:S95–S105, 2014.

[29] Y. Chabot, A. Bertaux, C. Nicolle, and T. Kechadi. Automatic Timeline Construction and Analysis for Computer Forensics Purposes. In *2014 IEEE Joint Intelligence and Security Informatics Conference*, pages 276–279, 2014.

[30] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 27–40. IEEE, 2001.

[31] C. C.-C. Cheng, C. Shi, N. Z. Gong, and Y. Guan. Logextractor: Extracting digital evidence from Android log messages via string and taint analysis. *Forensic Science International: Digital Investigation*, 37:301193, 2021.

[32] Y. Cheng, X. Fu, X. Du, B. Luo, and M. Guizani. A lightweight live memory forensic approach based on hardware virtualization. *Information Sciences*, 379:23–41, 2017.

[33] M. Choudhary and B. Kishore. Haamd: Hybrid analysis for Android malware detection. In *2018 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4. IEEE, 2018.

[34] M. Chua and V. Balachandran. Effectiveness of Android obfuscation on evading anti-malware. In *Proceedings of the eighth ACM conference on data and application security and privacy*, pages 143–145, 2018.

[35] M. I. Cohen, D. Bilby, and G. Caronni. Distributed forensics and incident response in the enterprise. *digital investigation*, 8:S101–S110, 2011.

[36] J. Cosic and M. Baca. A Framework to (Im) Prove "Chain of Custody" in Digital Investigation Process. In *CECIIS*,

page 435, 2010.

[37] E. Cunningham. Keeping you safe with google play protect. *Retrieved March*, 28:2020, 2017.

[38] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[39] S. Dai, T. Wei, and W. Zou. DroidLogger: Reveal suspicious behavior of Android applications via instrumentation. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pages 550–555, 2012.

[40] M. Debinski, F. Breitinger, and P. Mohan. Timeline2GUI: A Log2Timeline CSV parser and training scenarios. *Digital Investigation*, 28:34–43, 2019.

[41] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red-handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 15–27, 2019.

[42] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis. REAPER: real-time app analysis for augmenting the Android permission system. In *ACM CODASPY*, pages 37–48, 2019.

[43] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android hacker's handbook*. John Wiley & Sons, 2014.

[44] N. Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

[45] L. Englbrecht and G. Pernul. A privacy-aware digital forensics investigation in enterprises. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[46] Eventbot: A new mobile banking trojan is born. `https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born`, 2020. Accessed: 24.10.2020.

[47] H. Falaki, R. Mahajan, and D. Estrin. SystemSens: a tool for monitoring usage in smartphone research deployments. In *Proceedings of the sixth international workshop on MobiArch*, pages 25–30, 2011.

[48] P. Feng, Q. Li, P. Zhang, and Z. Chen. Private data acquisition method based on system-level data migration and volatile memory forensics for Android applications. *IEEE Access*, 7:16695–16703, 2019.

[49] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In *IEEE S&P*, pages 1041–1057. IEEE, 2017.

[50] FRIDA. `https://frida.re/`. Accessed: 6.12.2019.

[51] J. D. Gibbons and S. Chakraborti. *Nonparametric statistical inference*. CRC press, 2020.

[52] Google. File-Based Encryption. `https://source.android.com/security/encryption/file-based` Accessed: 24.03.2021.

[53] Google. Timesketch: forensic timeline analysis. `https://github.com/google/timesketch` Accessed: 24.03.2021.

[54] What's new with BigQuery ML: Unsupervised anomaly detection for time series and non-time series data. `https://cloud.google.com/blog/products/data-analytics/bigquery-ml-unsupervised-anomaly-detection`, July 2021. Accessed: 15.04.2023.

[55] J. Grover. Android forensics: Automated data collection and reporting from a mobile device. *Digital Investigation*, 10:S12–S20, 2013.

[56] K. Guðjónsson. Mastering the super timeline with log2timeline. *SANS Institute*, 2010. Accessed: 8.4.2021.

[57] H. Guo, S. Yuan, and X. Wu. Logbert: Log anomaly detection via bert. In *2021 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2021.

[58] Gustuff: Weapon of Mass Infection. `https://www.group-ib.com/blog/gustuff`, 2019. Accessed: 24.03.2021.

[59] C. Hargreaves and J. Patterson. An automated timeline reconstruction approach for digital forensic investigations. *Digital Investigation*, 9:S69–S79, 2012.

[60] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller. Honey, I shrunk your app security: The state of android app hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, pages 69–91. Springer, 2018.

[61] X. Hei, X. Du, and S. Lin. Two vulnerabilities in Android OS kernel. In *2013 IEEE International Conference on Communications (ICC)*, pages 6123–6127, 2013.

[62] J. Hizver and T.-c. Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 3–14, 2014.

[63] A. Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier, 2011.

[64] K. T. Kalleberg and O. A. V. Ravnås. Testing interoperability with closed-source software through scriptable diplomacy. In *FOSDEM*, 2016.

[65] G. Kim, S. Kim, M. Park, Y. Park, I. Lee, and J. Kim. Forensic analysis of instant messaging apps: Decrypting Wickr and private text messaging data. *Forensic Science International: Digital Investigation*, 37:301138, 2021.

[66] P. Kotzias, J. Caballero, and L. Bilge. How did that get in my phone? unwanted app distribution on Android devices. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 53–69. IEEE, 2021.

[67] A. Kuppa, S. Grzonkowski, M. R. Asghar, and N.-A. Le-Khac. Finding rats in cats: Detecting stealthy attacks using group anomaly detection. In *2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, pages 442–449. IEEE, 2019.

[68] M. La Polla, F. Martinelli, and D. Sgandurra. A Survey on Security for Mobile Devices. *IEEE Communications Surveys & Tutorials*, 15(1):446–471, 2013.

[69] H. Lee, H. Moon, I. Heo, D. Jang, J. Jang, K. Kim, Y. Paek, and B. B. Kang. Ki-mon arm: A hardware-assisted event-triggered monitoring platform for mutable kernel object. *IEEE Transactions on Dependable and Secure Computing*, 16(2):287–300, 2017.

[70] Y. Leguesse, M. Vella, C. Colombo, and J. Hernandez-Castro. Reducing the forensic footprint with Android accessibility attacks. In *International Workshop on Security and Trust Management, EUSORICS Conference*, pages 22–38. Springer, 2020.

[71] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 829–844, 2017.

[72] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. John Wiley & Sons, 2014.

[73] Z. Liu, T. Qin, X. Guan, H. Jiang, and C. Wang. An Integrated Method for Anomaly Detection From Massive System Logs. *IEEE Access*, 6:30602–30611, 2018.

[74] J. Lopes, C. Serrão, L. Nunes, A. Almeida, and J. Oliveira. Overview of machine learning methods for Android malware identification. In *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6. IEEE, 2019.

[75] J. T. Luttgens, M. Pepe, and K. Mandia. *Incident Response & Computer Forensics*. McGraw-Hill Education Group, 3rd edition, 2014.

[76] H. H. Lwin, W. P. Aung, and K. K. Lin. Comparative Analysis of Android Mobile Forensics Tools. In *2020 IEEE*

Conference on Computer Applications(ICCA), pages 1–6, 2020.

[77] A. Mehtab, W. B. Shahid, T. Yaqoob, M. F. Amjad, H. Abbas, H. Afzal, and M. N. Saqib. AdDroid: rule-based machine learning framework for Android malware analysis. *Mobile Networks and Applications*, 25:180–192, 2020.

[78] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame. You Shall not Repackage! Demystifying Anti-Repackaging on Android. *Computers & Security*, page 102181, 2021.

[79] K. Mistele. A Beginner's Guide to EDR Evasion. `https://kylemistele.medium.com/a-beginners-guide-to-edr-evasion-b98cc076eb9a`, 2021. Accessed: 09.07.2023.

[80] J. Mohamad Arif, M. F. Ab Razak, S. Awang, S. R. Tuan Mat, N. S. N. Ismail, and A. Firdaus. A static analysis approach for Android permission-based malware detection systems. *PloS one*, 16(9):e0257968, 2021.

[81] M. Mohamed, B. Shrestha, and N. Saxena. Smashed: Sniffing and manipulating Android sensor data for offensive purposes. *IEEE Transactions on Information Forensics and Security*, 12(4):901–913, 2016.

[82] M. Muscat and M. Vella. Enhancing virtual machine introspection-based memory analysis with event triggers. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 133–136. IEEE, 2018.

[83] G. Na, J. Lim, S. Lee, and J. Yi. Mobile Code Anti-Reversing Scheme Based on Bytecode Trapping in ART. *Sensors*, 19:2625, 06 2019.

[84] T. P. Android - Statistics & Facts. `https://www.statista.com/topics/876/android/`, 2023. Accessed: 04.03.2023.

[85] F. Pagani. *Advances in Memory Forensics*. PhD thesis, Sorbonne université, 2019.

[86] F. Pagani and D. Balzarotti. Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security*, 25(1):1–26, 2021.

[87] F. Pagani, O. Fedorov, and D. Balzarotti. Introducing the temporal dimension to memory forensics. *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–21, 2019.

[88] J. Park, Y.-H. Jang, and Y. Park. New flash memory acquisition methods based on firmware update protocols for LG Android smartphones. *Digital Investigation*, 25:42–54, 2018.

[89] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, pages 583–595, 2016.

[90] J. Pescatore. SANS top new attacks and threat report. *SANS Institute*, 2019.

[91] Fake WhatsApp app downloaded more than one million times. `https://www.bbc.com/news/technology-41886157`, 2017. Accessed: 16.08.2022.

[92] Researchers Flag 300K Banking Trojan Infections from Google Play in 4 Months. `https://threatpost.com/banking-trojan-infections-google-play/176630/`, 2021. Accessed: 16.08.2022.

[93] New Android malware on Google Play installed 3 million times. `bleepingcomputer.com/news/security/new-android-malware-on-google-play-installed-3-million-times/`, 2022. Accessed: 05.02.2023.

[94] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[95] Z. Qi, Y. Qu, and H. Yin. Logicmem: Automatic profile generation for binary-only memory forensics via logic inference. In *Proceedings 2022 Network and Distributed System Security Symposium. Internet Society*, 2022.

[96] J. Qin, H. Zhang, J. Guo, S. Wang, Q. Wen, and Y. Shi. Vulnerability Detection on Android Apps - Inspired by Case Study on Vulnerability Related With Web Functions. *IEEE Access*, 8:106437–106451, 2020.

[97] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–426, 2017.

[98] C. E. Rubio-Medrano, P. K. D. Soundrapandian, M. Hill, L. Claramunt, J. Baek, and G.-J. Ahn. DyPolDroid: Protecting against permission-abuse attacks in Android. *Information Systems Frontiers*, 25(2):529–548, 2023.

[99] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo. Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization. In *Annual Computer Security Applications Conference*, pages 970–981, 2021.

[100] A. Ruggia, A. Possemato, A. Merlo, D. Nisi, and S. Aonzo. Android, notify me when it is time to go phishing. In IEEE, editor, *EURO S&P 2023, 8th IEEE European Symposium on Security and Privacy, 3-7 July 2023, Delft, Netherlands*, Delft, 2023.

[101] F. E. Salamh, U. Karabiyik, and M. K. Rogers. Asynchronous Forensic Investigative Approach to Recover Deleted Data from Instant Messaging Applications. In *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2020.

[102] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. GUITAR: Piecing together Android app GUIs from memory images. In *CCS*, pages 120–132. ACM SIGSAC, 2015.

[103] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. VCR: App-agnostic recovery of photographic evidence from Android device memory images. In *CCS*, pages 146–157, 2015.

[104] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III. Screen after previous screens: Spatial-temporal recreation of Android app displays from memory images. In *USENIX*, pages 1137–1151, 2016.

[105] B. D. Saltaformaggio. *Convicted by memory: Automatically recovering spatial-temporal evidence from memory images*. PhD thesis, Purdue University, 2016.

[106] D. Sazonov. Andriller - Android Forensic Tools. `https://github.com/den4uklandriller`. Accessed: 18.02.2023.

[107] J. Schutte, D. Titze, and J. M. De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into Android apps. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 370–379. IEEE, 2014.

[108] S. Sentanoe and H. P. Reiser. SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic. *Forensic Science International: Digital Investigation*, 40:301337, 2022.

[109] D. Shen, Z. Zhang, X. Ding, Z. Li, and R. Deng. H-binder: A hardened binder framework on Android systems. In *Security and Privacy in Communication Networks: 12th International Conference, SecureComm 2016, Guangzhou, China, October 10-12, 2016, Proceedings 12*, pages 24–43. Springer, 2017.

[110] L. Shi, J. Fu, Z. Guo, and J. Ming. Jekyll and Hyde is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *MobiSys*, pages 222–235. ACM, 2019.

[111] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan. Vahunt: Warding off new repackaged Android malware in app-virtualization's clothing. In *ACM SIGSAC*, pages 535–549, 2020.

[112] R. Spolaor, E. Dal Santo, and M. Conti. Delta: Data extraction and logging tool for Android. *IEEE Transactions on Mobile Computing*, 17(6):1289–1302, 2017.

[113] H. Srivastava and S. Tapaswi. Logical acquisition and analysis of data from Android mobile devices. *Information & Computer Security*, 2015.

[114] M. Stone. Bad Binder: Android In-The-Wild Exploit. `https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html`, November 2019. Accessed: 07.07.2023.

[115] H. Studiawan and F. Sohel. Anomaly detection in a forensic timeline with deep autoencoders. *Journal of Information*

Security and Applications, 63:103002, 2021.

[116] H. Studiawan, F. Sohel, and C. Payne. Sentiment Analysis in a Forensic Timeline With Deep Learning. *IEEE Access*, 8:60664–60675, 2020.

[117] X. Su, L. Xiao, W. Li, X. Liu, K.-C. Li, and W. Liang. DroidPortrait: Android malware portrait construction based on multidimensional behavior analysis. *Applied Sciences*, 10(11):3978, 2020.

[118] S. Sudhakaran, A. Ali-Gombe, A. Orgah, A. Case, and G. G. Richard. AmpleDroid recovering large object files from Android application memory. In *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6. IEEE, 2020.

[119] J. Sylve. Lime-linux memory extractor. In *Proceedings of the 7th ShmooCon conference*, 2012.

[120] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of Android malware behaviors. In *Ndss*, pages 1–15, 2015.

[121] R. Tamma and D. Tindall. *Learning Android forensics*. Packt Publishing Ltd, 2015.

[122] B. Taubmann, O. Alabduljaleel, and H. Reiser. DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps. *Digital Investigation*, 26:S67–S76, 2018.

[123] B. Taubmann, C. Frädrich, D. Dusold, and H. P. Reiser. TLSkex: Harnessing virtual machine introspection for decrypting TLS communication. *Digital Investigation*, 16:S114–S123, 2016.

[124] D. Team. DroidPlugin. `https://github.com/DroidPluginTeam/DroidPlugin`, December 2020. Accessed: 03.12.2021.

[125] R. Team. Rekall memory forensic framework: about the Rekall memory forensic framework. *Retrieved March*, 13:2015, 2015.

[126] M. Toslali, E. Ates, A. Ellis, Z. Zhang, D. Huye, L. Liu, S. Puterman, A. K. Coskun, and R. R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *ACM SoCC*, pages 61–75, 2021.

[127] Triada: organized crime on Android. `https://www.kaspersky.com/blog/triada-trojan/11481/`, 2016. Accessed: 18.02.2022.

[128] R. Umar, I. Riadi, and G. M. Zamroni. A Comparative Study of Forensic Tools for WhatsApp Analysis using NIST Measurements. *Int. J. Adv. Comput. Sci. Appl*, 8(12):69–75, 2017.

[129] R. Umar, I. Riadi, G. M. Zamroni, et al. Mobile forensic tools evaluation for digital crime investigation. *Int. J. Adv. Sci. Eng. Inf. Technol*, 8(3):949, 2018.

[130] M. Vella and V. Rudramurthy. Volatile memory-centric investigation of SMS-hijacked phones: a Pushbullet case study. In *FedCSIS*, pages 607–616. IEEE, 2018.

[131] Jining Luohe Network Technology Co. Ltd. 2020. VirtualApp. `https://github.com/asLody/VirtualApp`, December 2021. Accessed: 03.12.2021.

[132] Volatility Framework. `https://www.volatilityfoundation.org/` Accessed: 10.07.2023.

[133] P. Wächter and M. Gruhn. Practicability study of Android volatile memory forensic research. In *2015 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6, 2015.

[134] D. T. Wagner, A. Rice, and A. R. Beresford. Device analyzer: Understanding smartphone usage. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services: 10th International Conference, MOBIQUITOUS 2013, Tokyo, Japan, December 2-4, 2013, Revised Selected Papers 10*, pages 195–208. Springer, 2014.

[135] Q. Wang, X. Zhang, X. Wang, and Z. Cao. Log sequence anomaly detection method based on contrastive adversarial training and dual feature extraction. *Entropy*, 24(1):69, 2021.

[136] S. Wang, Y. Wang, X. Zhan, Y. Wang, Y. Liu, X. Luo, and S.-C. Cheung. Aper: evolution-aware runtime permission misuse detection for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*, pages 125–137, 2022.

[137] X. Wang, S. Shi, Y. Chen, and W. C. Lau. PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[138] Whittaker, Zack. Eventbot: A new mobile banking trojan is born. `https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born`, 2020. Accessed: 24.03.2021.

[139] Understanding the Xposed Framework for ART Runtime. `https://mssun.me/blog/understanding-xposed-framework-art-runtime.html`, 2015. Accessed: 01.08.2022.

[140] MSAB XRY. `https://www.msab.com/wp-content/uploads/2021/11/XRY_Logical_EN.pdf`, 2021. Accessed: 8.11.2021.

[141] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 289–306, 2017.

[142] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang. Auditing anti-malware tools by evolving Android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, 2017.

[143] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX security symposium (USENIX security 12)*, pages 569–584, 2012.

[144] H. Yang, J. Zhuge, H. Liu, and W. Liu. A tool for volatile memory acquisition from android devices. In *Advances in Digital Forensics XII: 12th IFIP WG 11.9 International Conference, New Delhi, January 4-6, 2016, Revised Selected Papers 12*, pages 365–378. Springer, 2016.

[145] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen. UIScope: Accurate, Instrumentation-free, and Visible Attack Investigation for GUI Applications. In *Network and Distributed Systems Security (NDSS)*, 2020.

[146] S. J. Yang, J. H. Choi, K. B. Kim, R. Bhatia, B. Saltaformaggio, and D. Xu. Live acquisition of main memory data from Android smartphones and smartwatches. *Digital Investigation*, 23:50–62, 2017.

[147] S. J. Yang, J. H. Choi, K. B. Kim, and T. Chang. New acquisition method based on firmware update protocols for Android smartphones. *Digital Investigation*, 14:S68–S76, 2015.

[148] Y. Yang, Y. Zhang, and Z. Lin. Cross Miniapp Request Forgery: Root Causes, Attacks, and Vulnerability Detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3079–3092, 2022.

[149] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices. *IEEE Transactions on Dependable and Secure Computing*, 17(1):209–222, 2017.

[150] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *USENIX OSDI*, pages 293–306, 2012.

[151] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM TOCS*, 30(1):1–28, 2012.

[152] R. Zhang, M. Xie, and J. Bian. ReLF: Scalable Remote Live Forensics for Android. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 822–831. IEEE, 2021.

[153] X. Zhang, I. Baggili, and F. Breitinger. Breaking into the vault: Privacy, security and forensic analysis of Android vault applications. *Computers & Security*, 70:516–531, 2017.

[154] X. Zhang, F. Breitinger, E. Luechinger, and S. O'Shaughnessy. Android application forensics: A survey of obfus-

cation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.

[155] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *ACM SOSP*, pages 565–581, 2017.

[156] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: supporting and enforcing security profiles on smartphones. *IEEE TDSC*, 11(3):211–223, 2014.

A JIT-MF Drivers

```
Driver_ID: TG_CP
Scope: <telegram, crime-proxy>

/*Attributes*/
Evidence_objects: {<"Telegram Message Sent","org.telegram.messenger.MessageObject",
    carve_message_object(),parse_message_object(), {"1"}>}
Collection_method: online
Parsing_method: online
Triggers: {<"1",<"send",native, trigger_predicate(), trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/*Exposed interface*/
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/*Internal functions*/
bool trigger_predicate(params) {
  file_descriptor = params[1];
  if file_descriptor type is tcp:
    return true;
  else:
    return false;
}
void trigger_callback(thread_context) {
  /*the native function <send> takes a file
  descriptor as its only parameter*/
  if trigger_predicate(thread_context.args) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
```

```
    call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_message_object(from: address, to: address) {
  carve MessageObject in the given memory range using metadata provided by the
    Garbage Collector;
}

@OFFLINE
[object,...] carve_message_object_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_message_object(at) {
  if Parsing_method == online:
    current_time = get_time();
    MessageObject = object starting from at;

    message_content = MessageObject.messageText.value;
    message_date = MessageObject.messageOwner.date;
    message_id = MessageObject.messageOwner.id;

    append_log(Log_location,"{'time': current_time, 'event': Evidence_objects[0][0],
     'trigger_point':Triggers[0][0], 'object':{'date':message_date, 'message_id':
    message_id, 'text':message_content,");

    dump_rt_object(["org.telegram.messenger.MessageControllerObject",
    carve_message_controller_object(), parse_message_controller_object()]);

    return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>>];
  else:
    parse_message_object_offline(at);
}

@OFFLINE
[<field,value>,...] parse_message_object_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}

[object,...] carve_message_controller_object(from: address, to: address) {
  carve MessageControllerObject in the given memory range using metadata provided by
    the Garbage Collector;
}

@OFFLINE
[object,...] carve_message_object_controller_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_message_controller_object(at) {
  if Parsing_method == online:
    MessageControllerObject = object starting from at;
    recipient_id = MessageControllerObject.getUser();
    recipient_name = to_user.username.value;
    recipient_phone = to_user.phone.value;

    sender_id = device_owner;
    sender_name = device_owner;
    sender_phone_number = device_owner;

    append_log(Log_location, "'to_id':recipient_id, 'to_name':recipient_name, '
    to_phone':recipient_phone_number, 'from_id':sender_id, 'from_name':sender_name,
    'from_phone':sender_phone_number}}")

    return [<'to_id',recipient_id>, <'to_name',recipient_name>, <'to_phone',
    recipient_phone_number>, <'from_id',sender_id>, <'from_name',sender_name>, <'
    from_phone',sender_phone_number>];
  else:
    parse_message_controller_object_offline(at);
}

@OFFLINE
[<field,value>,...] parse_message_controller_object_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}

bool sampling_predicate(thread_context) {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing A.1: JIT-MF Driver for Section 5.3 Case Study A: Telegram Crime-Proxy.

```
Driver_ID: SIGNAL_CP
Scope: <signal, crime-proxy>
```

```

/*Attributes*/
Evidence_objects: {<"Signal Message Sent","org.thoughtcrime.securesms.conversation.
    ConversationMessage", carve_conversation_message(),parse_conversation_message(),
     {"1"}>}
Collection_method: online
Parsing_method: online
Triggers: {<"1",<"write",native, trigger_predicate(), trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/*Exposed interface*/
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/*Internal functions*/
bool trigger_predicate(params) {
  return true;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_conversation_message(from: address, to: address) {
  carve ConversationMessage in the given memory range using metadata provided by the
     Garbage Collector;
}

@OFFLINE
[object,...] carve_conversation_message_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_conversation_message(at) {
  if Parsing_method == online:
    current_time = get_time();
    ConversationMessage = object starting from at;

    MessageRecord = ConversationMessage.messageRecord;

    message_date = MessageRecord.dateSent.value;
    message_id = MessageRecord.id.value;
    message_content =  MessageRecord.body.value;

    if MessageRecord.isOutgoing():
      recipient_id = messageRecord.individualRecipient.id.value;
      recipient_name = messageRecord.individualRecipient.username.value;
      recipient_phone = messageRecord.individualRecipient.e164.value;

      sender_id = owner Signal ID;
      sender_name = owner Signal username;
      sender_phone_number = owner phone number;
    else
      recipient_id = owner Signal ID;
      recipient_name = owner Signal username;
      recipient_phone = owner phone number;

      sender_id = messageRecord.individualRecipient.id.value;
      sender_name = messageRecord.individualRecipient.username.value;
      sender_phone_number = messageRecord.individualRecipient.e164.value;


    return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>,<'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
  else:
    parse_conversation_message_offline(at);
}

@OFFLINE
[<field,value>,...] parse_conversation_message_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}

bool sampling_predicate(thread_context) {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/ * Helper  function * /
datetime get_time(){
  return current time;
}
```

Listing A.2: JIT-MF Driver for Section 5.3 Case Study B: Signal Crime-Proxy.

```
Driver_ID: PUSHBULLET_CP
Scope: <pushbullet, crime-proxy>

/* Attributes */
Evidence_objects: {<"Pushbullet Message Sent","org.json.JSONObject",
    carve_json_object(),parse_json_object(), {"1"}>}
Collection_method: online
Parsing_method: offline
Triggers: {<"1",<"write",native, trigger_predicate(), trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/* Exposed interface */
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3],Processes);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/* Internal functions */
bool trigger_predicate(params) {
  return true;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_json_object(from: address, to: address) {
  carve JSONObject in the given memory range using metadata provided by the Garbage
    Collector;
}

@OFFLINE
[object,...] carve_json_object_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}
```

```

[<field,value>,...] parse_json_object(at) {
  if Parsing_method == online:
  //  parse object fields starting at the given address;
  else:
    parse_json_object_offline(at);

}

@OFFLINE
[<field,value>,...] parse_json_object_offline(at) {
  if Collection_method == online:
    current_time = get_time();
    JSONObject = object starting from at;

    str1='{"active":.*"message":.*}}';

    res1 = regex match for str1 in JSONObject.toString();

    if(res1!==null){

      obj = JSON.parse(res1);
      message_date = obj.data.timestamp;
      message_id = obj.iden;
      message_content = obj.data.message;

      recipient_phone_number = obj.data.addresses[0] ;
      recipient_id = "";
      recipient_name = "";

      if (obj.data.status == "sent") {
        sender_phone_number = owner phone number;
        sender_id = "";
        sender_name = owner name;
      }
      object = '{"date": "' + date + '", "message_id": "' + msg_id + '", "text": "'
    + text + '", "to_id": "", "to_name": "", "to_phone": "' + to_phone  + '", "
    from_id": "", "from_name": "", "from_phone": "' + from_phone  + '"}';

      append_log(Log_location,"{'time': current_time, 'event': Evidence_objects
    [0][0], 'trigger_point':Triggers[0][0], 'object':{'date':message_date, '
    message_id':message_id, 'text':message_content,'to_id':recipient_id, 'to_name':
    recipient_name, 'to_phone':recipient_phone_number, 'from_id':sender_id, '
    from_name':sender_name, 'from_phone':sender_phone_number}}");

      return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>,<'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
  else:
    // use an hprof parser to parse object_j at the given offset from memory dump;
  }
}

bool sampling_predicate(thread_context) {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing A.3: JIT-MF Driver for Section 5.3 Case Study C: Pushbullet Crime-Proxy.

```
Driver_ID: TG_SP
Scope: <telegram, spying>

/* Attributes */
Evidence_objects: {<"Telegram Message Intercepted","org.telegram.messenger.
    MessageObject", carve_message_object(),parse_message_object(), {"1"}>}
Collection_method: online
Parsing_method: online
Triggers: {<"1",<"recv",native, trigger_predicate(), trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/* Exposed interface */
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/* Internal functions */
bool trigger_predicate(params) {
  file_descriptor = params[1];
  if file_descriptor type is tcp:
    return true;
  else:
    return false;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context.args) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_message_object(from: address, to: address) {
  carve MessageObject in the given memory range using metadata provided by the
    Garbage Collector;
}

@OFFLINE
[object,...] carve_message_object_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_message_object(at) {
  if Parsing_method == online:
    current_time = get_time();
    MessageObject = object starting from at;

    message_content = MessageObject.messageText.value;
    message_date = MessageObject.messageOwner.date;
    message_id = MessageObject.messageOwner.id;

    append_log(Log_location,"{'time': current_time, 'event': Evidence_objects[0][0],
     'trigger_point':Triggers[0][0], 'object':{'date':message_date, 'message_id':
    message_id, 'text':message_content,");

    dump_rt_object(["org.telegram.messenger.MessageControllerObject",
    carve_message_controller_object(), parse_message_controller_object()]);

    return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>>];
  else:
    parse_message_object_offline(at);
}

@OFFLINE
[<field,value>,...] parse_message_object_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}
[object,...] carve_message_controller_object(from: address, to: address) {
  carve MessageControllerObject in the given memory range using metadata provided by
     the Garbage Collector;
}

@OFFLINE
[object,...] carve_message_object_controller_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_message_controller_object(at) {
  if Parsing_method == online:
    MessageControllerObject = object starting from at;
    sender_id = MessageControllerObject.getUser();
    sender_name = to_user.username.value;
    sender_phone_number = to_user.phone.value;

    recipient_id  = device_owner;
    recipient_name  = device_owner;
    recipient_phone  = device_owner;

    append_log(Log_location, "'to_id':recipient_id, 'to_name':recipient_name, '
    to_phone':recipient_phone_number, 'from_id':sender_id, 'from_name':sender_name,
    'from_phone':sender_phone_number}}")

    return [<'to_id',recipient_id>, <'to_name',recipient_name>, <'to_phone',
    recipient_phone_number>, <'from_id',sender_id>, <'from_name',sender_name>, <'
    from_phone',sender_phone_number>];
  else:
    parse_message_controller_object_offline(at);

}

@OFFLINE
[<field,value>,...] parse_message_controller_object_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}

bool sampling_predicate() {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/* Helper function */
```

```
datetime get_time(thread_context){
  return current time;
}
```

Listing A.4: JIT-MF Driver for Section 5.3 Case Study D: Telegram Spying.

```
Driver_ID: SIGNAL_SP
Scope: <signal, spying>

/* Attributes */
Evidence_objects: {<"Signal Message Intercepted","org.thoughtcrime.securesms.
    conversation.ConversationMessage", carve_conversation_message(),
    parse_conversation_message(), {"1"}>}
Collection_method: online
Parsing_method: online
Triggers: {<"1",<"open",native, trigger_predicate(), trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/* Exposed interface */
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/* Internal functions */
bool trigger_predicate(params) {
  return true;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_conversation_message(from: address, to: address) {
  carve ConversationMessage in the given memory range using metadata provided by the
     Garbage Collector;
}

@OFFLINE
[object,...] carve_conversation_message_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_conversation_message(at) {
  if Parsing_method == online:
    current_time = get_time();
    ConversationMessage = object starting from at;

    MessageRecord = ConversationMessage.messageRecord;

    message_date = MessageRecord.dateSent.value;
    message_id = MessageRecord.id.value;
    message_content =  MessageRecord.body.value;

    if MessageRecord.isOutgoing():
      recipient_id = messageRecord.individualRecipient.id.value;
      recipient_name = messageRecord.individualRecipient.username.value;
      recipient_phone = messageRecord.individualRecipient.e164.value;

      sender_id = owner Signal ID;
      sender_name = owner Signal username;
      sender_phone_number = owner phone number;
    else
      recipient_id = owner Signal ID;
      recipient_name = owner Signal username;
      recipient_phone = owner phone number;

      sender_id = messageRecord.individualRecipient.id.value;
      sender_name = messageRecord.individualRecipient.username.value;
      sender_phone_number = messageRecord.individualRecipient.e164.value;

    append_log(Log_location,"{'time': current_time, 'event': Evidence_objects[0][0],
     'trigger_point':Triggers[0][0], 'object':{'date':message_date, 'message_id':
    message_id, 'text':message_content,'to_id':recipient_id, 'to_name':
    recipient_name, 'to_phone':recipient_phone_number, 'from_id':sender_id, '
    from_name':sender_name, 'from_phone':sender_phone_number}}");

    return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>,<'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
  else:
    parse_conversation_message_offline(at);
}

@OFFLINE
[<field,value>,...] parse_conversation_message_offline(at) {
  // if Collection_method == online:
  //   use custom parser to parse object_j at the given offset
  // else
  //   use an hprof parser to parse object_j at the given offset from memory dump;
}
```

```

bool sampling_predicate(thread_context) {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing A.5: JIT-MF Driver for Section 5.3 Case Study E: Signal Spying.

```
Driver_ID: PUSHBULLET_SP
Scope: <pushbullet, spying>

/* Attributes */
Evidence_objects: {<"Pushbullet Message Synced","org.json.JSONObject",
    carve_json_object(),parse_json_object(), {"1"}>}
Collection_method: online
Parsing_method: offline
Triggers: {<"1",<"android.content.Intent.createFromParcel",rt, trigger_predicate(),
    trigger_callback()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"

/* Exposed interface */
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3],Processes);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/* Internal functions */
bool trigger_predicate(params) {
  return true;
}
void trigger_callback(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = Evidence_objects[0];
      object_name = object[1];
      object_carve_callback_fn = object[2];
      object_parse_callback_fn = object[3];
      dump_rt_object(object_name,object_carve_callback_fn,object_parse_callback_fn);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

[object,...] carve_json_object(from: address, to: address) {
  carve JSONObject in the given memory range using metadata provided by the Garbage
    Collector;
}

@OFFLINE
[object,...] carve_json_object_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_json_object(at) {
  if Parsing_method == online:
  //  parse object fields starting at the given address;
  else:
    parse_json_object_offline(at);
}

@OFFLINE
[<field,value>,...] parse_json_object_offline(at) {
  if Collection_method == online:
    current_time = get_time();
    JSONObject = object starting from at;

    str1='{"type":"push",.*"push":{"type":"sms_changed","source_device_iden":.*]}}';

    res1 = regex match for str1 in JSONObject.toString();

    if(res1!==null){

      obj = JSON.parse(res1);
      message_date = "";
      message_id = "";
      message_content = "";

      recipient_phone_number = "";
      recipient_id = "";
      recipient_name = "";

      if (obj.data.status == "sent") {
        sender_phone_number = owner phone number;
        sender_id = "";
        sender_name = owner name;
      }

      append_log(Log_location,"{'time': current_time, 'event': Evidence_objects
    [0][0], 'trigger_point':Triggers[0][0], 'object':{'date':message_date, '
    message_id':message_id, 'text':message_content,'to_id':recipient_id, 'to_name':
    recipient_name, 'to_phone':recipient_phone_number, 'from_id':sender_id, '
    from_name':sender_name, 'from_phone':sender_phone_number}}");

      return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>,<'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
  else:
    // use an hprof parser to parse object_j at the given offset from memory dump;
  }
}

bool sampling_predicate(thread_context) {
  current_time = get_time();
  get current_second from current_time;

  if (current_second \% 5 == 0):
    return true;
  else:
    return false;
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing A.6: JIT-MF Driver for Section 5.3 Case Study F: Pushbullet Spying.

```
Driver_ID: WHATSAPP_MSG_HIJACK
Scope: <whatsapp, msg-hijack>

/* Attributes */
Collection_method: online
Parsing_method: offline
Triggers: {<"1",<"android.database.sqlite.SQLiteDatabase.insert",rt,
    trigger_predicate_insert(), trigger_callback_insert()>,<"2",<"android.database.
    sqlite.SQLiteDatabase.update",rt, trigger_predicate_update(),
    trigger_callback_update()>>}
Sampling_method: sampling_predicate()
Log_location: "/sdcard/jitmflogs"
Globals:{<timestamp,>}
Evidence_objects: {} // the evidence object is added at runtime, since it is an
    argument to the trigger point

/* Exposed interface */
bool init (config) {
  for entry in Triggers:
    if entry[1] == native:
      place_native_hook("libc.so", entry[0], entry[3]);
    else:
      place_rt_hook(entry[0], entry[3]);
}

/* Internal  functions */
bool trigger_predicate_insert(params) {
  return true;
}
void trigger_callback_insert(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      evidence_object = thread_context.args[2]
      Evidence_objects.add(<"Whatsapp Messaging Event", evidence_object,
    carve_object_type(), parse_object_type(), {1}>})
      dump_rt_object(evidence_object,carve_content_values,parse_content_values);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}

bool trigger_predicate_update(params) {
  return true;
}
void trigger_callback_update(thread_context) {
  if trigger_predicate(thread_context) && sampling_predicate(thread_context):
    if Collection_method == online:
      object = thread_context.args[1]
      carved_content_value_object = carve_content_values(evidence_object start
    address, object end address);
      parse_content_values_for_timestamp(carved_content_value_object start address);
    else:
      call_rt_function("android.os.Debug.dumpHprofData",[Log_location]);
}


[object,...] carve_content_values(from: address, to: address) {
  carve ContentValues in the given memory range using metadata provided by the
    Garbage Collector;
}

@OFFLINE
[object,...] carve_content_values_offline(from: address, to: address) {
  // use an hprof parser to carve object_j in the given memory range;
}

[<field,value>,...] parse_content_values(at) {
  if Parsing_method == online:
  //  parse object fields starting at the given address;
  else:
    parse_json_object_offline(at);
}

@OFFLINE
[<field,value>,...] parse_content_values_offline(at) {
  if Collection_method == online:
    current_time = get_time();
    ContentValuesObject = object starting from at;
    ContentValuesObject_strings = base64 decode strings in ContentValuesObject;

    message_content = last string in ContentValuesObject_strings;
    message_id = "";
    message_date = Globals["timestamp"];

    if 'SendE2EMessageJob' in ContentValuesObject_strings:
      recipient_phone_number = match for regex search "@s.whatsapp.netsr[0-9]+\\n"
    in ContentValuesObject_strings;
      recipient_id = "";
      recipient_name = "";

      sender_phone_number = owner phone number;
      sender_id = "";
      sender_name = owner name;
      event_additional_info = "sent";
    else if 'SendReadReceiptJob' in ContentValuesObject_strings:
      sender_phone_number = match for regex search "@s.whatsapp.netsr[0-9]+\\n" in
    ContentValuesObject_strings;
      sender_id = "";
      sender_name = "";

      recipient_phone_number = owner phone number;
      recipient_id = "";
      recipient_name = owner name;
      event_additional_info = "read";
    }

      append_log(Log_location,"{'time': current_time, 'event': Evidence_objects
    [0][0]+'-'event_additional_info, 'trigger_point':Triggers[0][0], 'object':{'date
    ':message_date, 'message_id':message_id, 'text':message_content,'to_id':
    recipient_id, 'to_name':recipient_name, 'to_phone':recipient_phone_number, '
    from_id':sender_id, 'from_name':sender_name, 'from_phone':sender_phone_number
    }}");

      return [<'time', current_time>, <'event', Evidence_objects[0][0]>, <'
    trigger_point',Triggers[0][0]>, <'object',<'date',message_date>, <'message_id',
    message_id>, <'text',message_content>,<'to_id',recipient_id>, <'to_name',
    recipient_name>, <'to_phone',recipient_phone_number>, <'from_id',sender_id>, <'
    from_name',sender_name>, <'from_phone',sender_phone_number>>];
  else:
    // use an hprof parser to parse object_j at the given offset from memory dump;
  }
}

[<field,value>,...] parse_content_values_for_timestamp(at) {
  ContentValuesObject = object starting from at;
  Globals["timestamp"] = ContentValuesObject["sort_timestamp"];
  return [<>];
}
```

```

bool sampling_predicate(thread_context) {
  return true;
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing A.7: JIT-MF Driver for WhatsApp Section 6.1 Case Study.