IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Responding to targeted stealthy attacks on Android using timely-captured memory dumps

JENNIFER BELLIZZI[1], MARK VELLA[1], CHRISTIAN COLOMBO[1], AND JULIO HERNANDEZ-CASTRO[2]

[1]Department of Computer Science, University of Malta, Msida, Malta {jennifer.bellizzi,mark.vella, christian.colombo}@um.edu.mt
[2]School of Computing, Cornwallis South, University of Kent, Canterbury, UK jch27@kent.ac.uk

Corresponding author: Jennifer Bellizzi (e-mail: jennifer.bellizzi@um.edu.mt).

This work was supported by the LOCARD Project under Grant H2020-SU-SEC-2018-832735.

ABSTRACT The increasing dominance of Android smartphones for everyday communication and data processing makes long-term stealthy malware an even more dangerous threat. Recent malware campaigns like Flubot demonstrate that by employing stealthy malware techniques even at minimal capacity, malware is highly effective in making its way to millions of devices with little resistance from existing detection mechanisms. Consequential late detection demands comprehensive forensic timelines to reconstruct all malicious activities. However, the reduced forensic footprint of stealthy attacks with minimal malware involvement leaves investigators little evidence to work with even when utilising state-of-the-art digital forensics tools. Volatile memory forensics can be effective in such scenarios since app execution of any form is always bound to leave a trail of evidence in memory, even if it is short-lived. In this work, we motivate the need for *JIT-MF* (Just-in-time Memory Forensics), a technique that aims to address the challenges that arise with timely collection of short-lived evidence in volatile memory to solve the stealthiest of Android attacks. By taking an incident-response-centric approach, focused on protecting the device users rather than treating them as potential adversaries, we show that JIT-MF tools can collect elusive attack steps in volatile memory without requiring device rooting. Furthermore, we build *MobFor*, a JIT-MF based tool focusing on capturing evidence related to messaging hijack attacks. This tool provides a context to explore solutions for JIT-MF implementation challenges, aiming to render JIT-MF tools practical for real-world requirements. Finally, we demonstrate that when compared to state-of-the-art digital forensic tools Belkasoft and XRY in a realistic attack scenario involving an enhanced version of the WhatsApp Pink malware and stock Android devices, only MobFor can recover the contents of messages sent by the malware, hence decisively contributing to an enriched forensic timeline.

INDEX TERMS Memory Forensics, Android Security, Digital Forensics, Incident Response, Forensic Timelines

I. INTRODUCTION

ANDROID has established itself as a leader in the mobile OS market [1], making devices both a rich source of evidence as well as their users a primary target for cyberattacks. Whereas several detection mechanisms exist in the Google Play Protect suite [2], both to hinder the availability of malicious apps as well as to provide on-device detection, evasion techniques like obfuscation and stealthy execution enable malware to evade this protection layer [3].

A. INCIDENT RESPONSE SCOPE

Our primary concern is Android malware that exhibits long-term stealth by evading early detection mechanisms, which eventually is only detected much later through its consequences. Established stealthy attack vectors, such as accessibility [4] or app-level virtualisation misuse [5], and several others [6]–[8], have become increasingly popular among malware authors, with many recent incidents gaining worldwide reach, leaving devastating effects [9]–[12]. More seriously for incident response, the resulting reduced

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

forensic footprint for any attack employing such attack vectors has also been demonstrated [13], [14]. Specifically, in this work, we focus on stealthy attacks which hijack the messaging functionality of benign apps on Android devices to hide compromising communication of a criminal nature behind victim devices or spy on target victims through unlawful interception of messages.

The FluBot malware campaign [12], for example, falls in this category, tricking users into installing it via malicious URL links inside SMS text messages, subsequently delegating its core attack steps of fetching contacts and forwarding the URL links through legitimate messaging apps. It then proceeds to steal the credentials belonging to other apps found on the same device. In doing so, the FluBot malware conceals its presence and can remain undetected for a long time. Moreover, any messaging activity disclosed during an eventual investigation could be misattributed to the victim messaging app rather than to FluBot, quickly derailing the incident response process as well. It becomes paramount to recreate the intrusion scenario in such cases by identifying the main attack steps to establish a rich forensic timeline. Moreover, it becomes critical that all evidence related to the attack gets collected since that elusive evidence that goes uncollected could prove to be the missing piece of the puzzle that leaves the incident unsolved.

When investigating such advanced stealthy incidents, forensic artefacts constituting attack evidence would have been erased from storage or never even created to begin in the first place. This is the reality that incident responders face amidst the limitations and barriers of the more forensically sound, but alas limited, state-of-the-art mobile forensics. Regardless of the stealthiness of an attack, its execution must occur in memory [15], [16]. Therefore, volatile memory forensics becomes crucial to recover key artefacts in memory related to possible stealthy attack steps. Recent efforts [17]–[20] focus on reconstructing critical data objects from memory to build a more comprehensive timeline. However, these works rely on obtaining a memory image of a running Android process. This means that the device needs to be somewhat customised (rooted, having an unlocked bootloader, or customised firmware). Furthermore, in the case of long-term stealthy attacks, ill-timed memory images may not always contain evidence of an attack that has occurred at some point in the past.

B. RESEARCH PROBLEM

To bridge these gaps, we present a thorough study of *JIT-MF* (Just-in-time Memory Forensics), a framework introduced and preliminarily explored in previous works [21], [22], to determine its effectiveness towards generating more comprehensive forensic timelines in the case of stealthy message hijack attacks. In contrast to other digital forensics tools, a JIT-MF tool comes into play at the forensic readiness stage [23] to forensically enhance Android devices before an incident is flagged. In this setting, it makes sense to assume that the targeted device owners are willing to collaborate with investigators since they present the potential cyber attack targets. Importantly, it is this same context that can give any JIT-MF forensics tool the potential to collect evidence missed by state-of-the-art mobile forensics tools [24]. These tools target the more classical context of delving through layers of protection to extract evidence that could compromise its owner's position with the law. JIT-MF tools, in contrast, aim to protect device owners from cybercriminals; therefore, investigators using JIT-MF tools can rely on device owners' collaboration. This is not to say that existing tools become useless for our context. Rather, we argue — and later demonstrate in a real-world setting — that by complementing state-of-the-art tools with JIT-MF tools, it becomes possible to first get to that evidence missed by existing tools and subsequently produce rich forensic timelines contributed by the entire toolkit to have attack steps stand out from normal device usage, thus raising suspicion.

JIT-MF leverages static and dynamic app instrumentation to enable live process memory forensics by timely dumping key artefacts from memory. Figure 1 shows the complete JIT-MF workflow. At the forensic readiness stage, targeted users along with their devices and apps are identified during an asset management exercise (step 1). These 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 with JIT-MF drivers (step 2) – custom specification responsible for establishing the points in time when memory dumps should be triggered (trigger points) and the heap/native memory areas/objects to be included. The forensic collection of memory dumps is triggered by specific app events defined in the JIT-MF drivers (step 3). These memory dumps contain either raw binaries from memory segments or readily carved and parsed objects along with tagged metadata, e.g. in JSON format. These two collection methods are offline and online, respectively, depending on when object carving is carried out. Once suspicious activity is noticed, with alerts possibly raised by the device owners themselves or by incident responders during routine checks, JIT-MF dumps can be merged with other forensic sources to produce a more comprehensive forensic timeline (steps 4 and 5).

Since specific attack scenarios may require the involvement of legal entities, any evidence generated must be processed and gathered in a format so that it is acceptable in court. Therefore, throughout the entire process, a chain of custody for digital evidence is kept as shown in Figure 1, outlining the steps that were carried out throughout each stage of the workflow, as is required for any evidence produced to be accepted by courts as valid [25].

C. CONTRIBUTIONS

While shown to be effective in its efforts [22], JIT-MF still faces several challenges, primarily in terms of its practicality

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

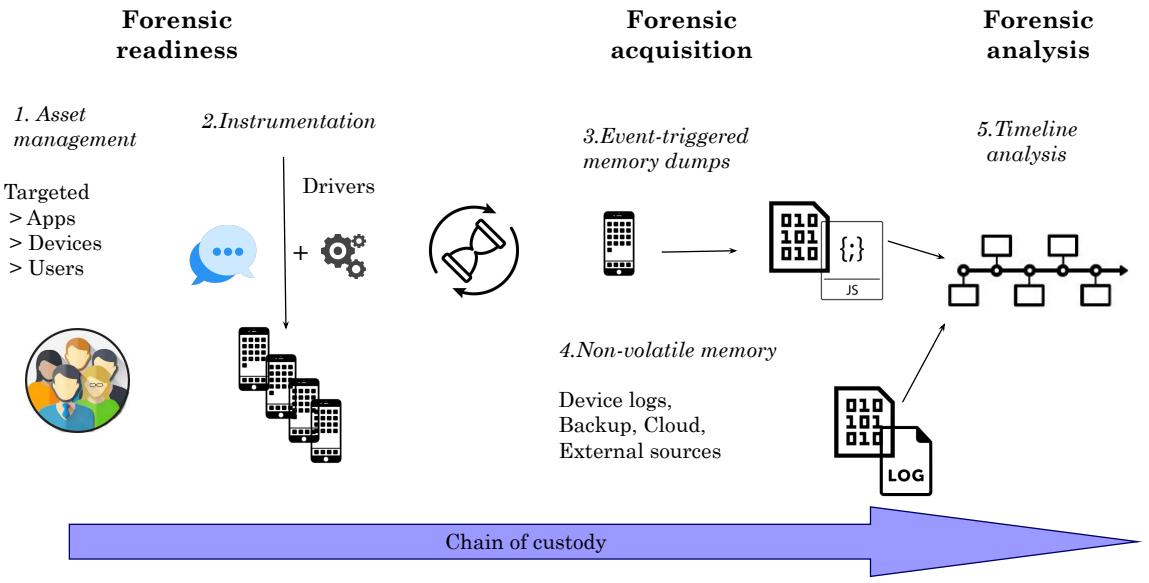


FIGURE 1: The JIT-MF workflow.

in a real-world setting. To address these challenges, we motivate the need for JIT-MF, explore it further through a sequence of experiments, as well as present MobFor, a JIT-MF incident response tool for messaging hijack attacks demonstrating the level of maturity of the proposed framework and enabling direct comparison with the state-of-the-art tools. Specifically, we make the following contributions, each addressing a specific *research question* (***RQ***):

- Does evidence of attack steps recovered through JIT-MF's real-time triggering of memory dumps enrich the corresponding timelines generated? (***RQ1***)
- Given a set of requirements for a JIT-MF-based tool, *MobFor*, what are the possible solutions that would render the tool practical? (***RQ2***)
- In what ways can *MobFor* complement existing forensic tools, in a realistic setting? (***RQ3***)

All of the above questions are answered within the scope of messaging hijack attack scenario, specifically four messaging apps are used throughout our experimentation spanning SMS (Pushbullet[1]) and Instant Messaging (Telegram[2], Signal[3], WhatsApp[4]). The first research question (***RQ1***) is fully answerable by previous work [22]. In this paper, we build on that existing work to create *MobFor*, which addresses the requirements presented by ***RQ2***, some of which have already been presented and partially answered in a preliminary study of JIT-MF [21]. Finally, through ***RQ3***, we assess the value that the additional forensic sources produced by *MobFor* contribute to existing standard digital forensics tools towards creating a more comprehensive forensic timeline in a realistic setting. We demonstrate this by recreating a criminal investigation with real malware targeting WhatsApp, across multiple commercially available non-rooted stock Android devices. Through the initial motivation for pursuing JIT-MF as a framework (***RQ1***), the creation of MobFor (***RQ2***) and the application of MobFor in a realistic setting (***RQ3***), we show that the JIT-MF framework has matured from a high-level technique to a practical tool that can be applied in real-world Android investigation scenarios.

The remainder of this paper is structured as follows. In the next section, we give an overview of the threat posed by stealthy Android malware, along with an overview of available digital forensics techniques to respond to them, as well as key techniques underpinning JIT-MF. Then, in Section III, we provide an overview of the related work, while in Section IV, we present and explain the fundamentals of the JIT-MF framework. In each of sections V, VI and VII, we address *RQ1, RQ2* and *RQ3*, respectively, and for each, describe the experiment setup, methodology and results obtained. Key takeaways from the results obtained and limitations and possible future work are discussed in Section VIII, and the conclusion is presented in the final section.

II. BACKGROUND

A. ANDROID ATTACK VECTORS FOR STEALTH

In recent years, several techniques have emerged which make for stealthier Android attack vectors by misusing legitimate actions with malicious intent to carry out attack-related events. This level of stealth makes it more difficult for malware detectors to do their job and detect such malware at runtime, leading to late detection and requiring incident response at a later stage.

[1]https://www.pushbullet.com/
[2]https://telegram.org/
[3]https://signal.org/
[4]https://www.whatsapp.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Android accessibility trojans are a case in point [10], [11]. Early instances [26] demonstrated how through phishing and the misuse of accessibility features, a malicious app could steal a victim's credentials and attack other 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. For instance, Eventbot [27] and BlackRock [28] malware only request the accessibility permission requested upon installation; the rest of the permissions required to perform the attack are obtained through the accessibility permission previously granted by the user. To increase the level of stealth even further, malware developers can also exploit accessibility to leverage critical benign app functionality that coincides with the features they need, as seen in Figure 2. In the case of message hijack attacks, attackers are interested in reading incoming messages (*spying*) or sending messages behind the victim's back (*sending* and *deleting* messages immediately). This functionality is no different from that typically offered by today's messaging (Figure 2a) or SMSonPC apps. Such apps enable sending text messages directly from the convenience of one's PC (Figure 2b), other than the fact that the initiator of these actions is a malicious actor and the device owner is unaware of these events. This attack vector has been shown to enable stealthy Living-Off-the-Land (LOtL) tactics [29], 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 [14]. Delegating an attack's core steps to benign apps has the consequence of bypassing malware detection mechanisms and making any follow-up response more challenging as reconstructing the attack steps distributed among trusted apps is non-straightforward.

Overall, any form of inter-app communication, whether for app functionality or testing purposes, can be misused similarly to avoid detection and complicate incident response. Cross-App WebView Infections (XAWI) [30], for instance, exploit legitimate Cross-App WebView navigation, 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. In the case of messaging, malicious apps can misuse this functionality to send messages via another benign app. Another example vector is SMASHeD [31], which exploits the Android debug bridge. It enables malicious apps, 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). Zygote and binder infection combined with a rooting exploit [6], as well as app-level virtualization frameworks [7] and third-party library infections [8] provide further attack vectors, potentially resulting in similarly stealthy attacks which render any efforts by classifier-based malware detectors futile and for which JIT-MF could be a solution in terms of incident response, provided a forensic readiness stage.

B. ANDROID FORENSICS

a: Forensic sources

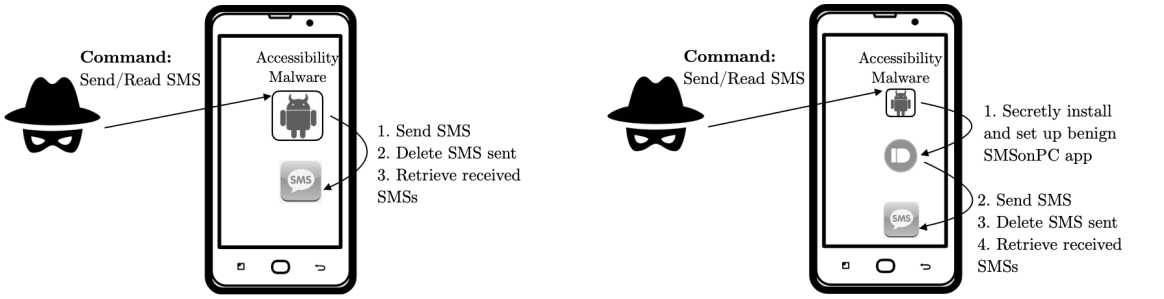
Android on-chip and removable flash memory constitute primary forensic sources, both device-wide and app-specific. System-wide sources can provide supplementary information about the underlying Linux kernel activities (via `dmesg`), system and device-wide app event logging (via `logcat`), user account audits, running services, device chipset info, cellular and Wi-Fi network activities (via `dumpsys`) [24]. The `/data/data` sub-tree of the Android filesystem (referring to internal storage where app persistent files and cache are stored) inside the `userdata` partition, along with the `sdcard` partition (referred to external storage where media is stored), is where it starts to get interesting, with app data typically stored in XML or SQLite files. 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 [32]). 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 [33], rendering it more practical and more 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 the collaboration of users or the use of exploits to decrypt the sources.

b: Collection methods, challenges, and analysis

Collection of forensic sources from Android devices can be carried out using logical or physical imaging [34]. Simply put, 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, as well as files saved to an app's internal storage, external storage, and in the database directory [35]. 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). All collection methods have to deal with Android's security barriers. For software-based collection,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access



(a) SMS hijack attack using accessibility to attack default SMS app.

(b) SMS hijack attack using accessibility to install an SMSonPC app that legitimately interacts with the default SMS app.

FIGURE 2: Misusing accessibility in different ways to carry out an SMS hijack attack.

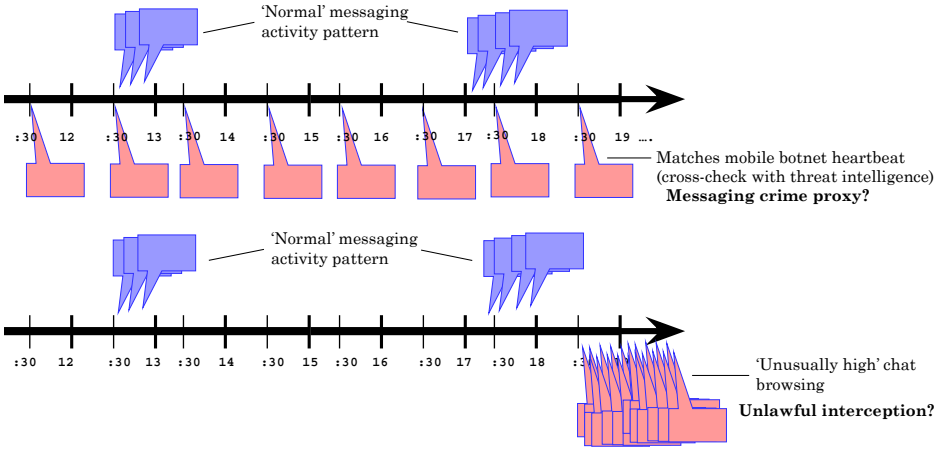


FIGURE 3: Forensic timelines supporting cyber-attack investigations.

the barriers range from locked screens, password-protected cloud storage to custom backup formats and rooting the device or downgrading the app to gain access to files stored in the app's internal storage [36]. While hardware-based/physical collections can bypass the above barriers, any form of physical imaging has to deal with FDE and FBE.

Rooting relies on exploiting some kernel or firmware flashing protocol vulnerability [34], [37], or else flashing a custom recovery partition through which to add some root-privileged utility. The latter may get further complicated by locked bootloaders. Restoring backups for forensic analysis is dependent on the type of backup taken [35]. 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 itself. Backups are typically stored 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.

The starting point for forensic analysis depends on what kind of collection is performed [24]. In the case of physical collection, it is necessary first 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.

c: Mobile forensics tools

Mobile forensics tools, e.g. Belkasoft [38] and MSAB's XRY [39], can target both the collection and the analysis of forensic sources from mobile devices. Each tool provides a set of collection options, equipped with rooting exploits and hardware interfacing cables, passcode brute-forcing methods as well as, in some cases, app downgrading options that are used to circumvent custom `BackupAgent` configurations that either encrypt or omit app data from backups. They also come equipped with parsing/analysis modules for file systems, databases, and app data formats. Ancillary analysis features, including timeline generation, provide a final pro-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

fessional touch to the product. Finally, evidence of interest can be exported in a format as required by the investigator, typically CSV or JSON format.

C. FORENSIC TIMELINES

Forensic timeline generation is widely considered to be the forensic analysis exercise that brings together all the collected evidence. It supports an investigator in reconstructing the hypothesised incident/crime scenario [40]. The richer the timelines, the greater the support for an investigator to reconstruct an intrusion/crime scene, thereby answering critical questions about an incident. For instance, in the case of a messaging hijack, see Figure 3 (top), the generated timeline can uncover a pattern matching a cyber threat intelligence feed for mobile botnet activity that could be leveraged for a crime messaging proxy. Else, see Figure 3 (bottom), timeline events can be compared to a baseline to identify unusual/suspicious activity. Both approaches can indicate the presence of an ongoing cyberattack, with a comprehensive timeline providing crucial support.

Many tools exist that target the generation and visualisation of super timelines to enable forensic analysis. Plaso [41] is a Python-based engine equipped with a range of tools that can be used for the automatic creation of super timelines. Timestamped data collected from several sources can be parsed and chronologically ordered in one super timeline using different Plaso parsers that are available for different application data. Furthermore, having publicly available source code, Plaso and its parsers can be further extended to cater to additional forensic sources as needed. Timesketch [42] is another open-source tool that enables forensic analysis. It takes as input preprocessed timeline data (e.g. generated by Plaso tools) in JSON (JavaScript Object Notation) or CSV (Comma Separated Value) format and outputs a visual timeline (called a *sketch*) that can be analysed. Furthermore, similar to Plaso, it can produce a single sketch using multiple inputs of preprocessed timeline data, containing parsed forensic evidence generated by various forensic analysis tools.

D. EVIDENCE COLLECTION FROM MEMORY

Android Runtime (ART) has been the main managed runtime used by applications on Android [43] since it was released with Android KitKat in 2013 [44]. Similar to how JVM operates, ART uses two separate memory spaces to store application data; the stack and the heap [45]. In a forensic investigation of an attack involving a benign target app, the data objects created during the runtime of that app, critical to attack steps, would constitute relevant forensic artefacts that need to be collected for further analysis. These critical data objects may only be found in volatile memory within the application's heap, managed by the Android Runtime. To collect these app objects from memory, one can either: i) Leverage ART functionality to solely dump the app's heap data, which requires making modifications to the app's codebase, or ii) Take an entire snapshot of the device's memory (i.e. a memory image), which requires making modifications to the device's underlying Android system.

Out of the box, ART provides functionality through which developers can dump their app's heap data in the standard format of an `hprof` file, mainly intended for debugging purposes but which can double as an important volatile memory forensics tool. `Debug.dumpHprofData`[5] is the Java API used to obtain a heap dump. A typical heap dump is semantically rich, containing information about an app's memory contents at the time when the dump was taken. Most importantly, in our case, it includes information on the objects used and created by the app [43]. Another ART feature of relevance to memory forensics is that of garbage collection [43]. Figure 4 shows how ART provides a managed memory environment which enables the Garbage Collector (GC) to keep track of objects in memory to reclaim heap space once those are no longer in use [43]. To do so, the GC uses a function exported by ART's binary module (`libart.so`), `Heap::GetInstances`[6], which has an object type filter, allowing the GC to filter on specific objects in memory. While convenient for selective evidence collection, the downside is that this function is not part of the public API and therefore may change unexpectedly between versions.

The acquisition of system memory images has been around for a while, with tools like Volatility [46] enabling it. However, this still poses several challenges on Android devices, primarily due to the sheer amount of different Android devices and manufacturers running different variants of the Android operating systems. Existing Android memory acquisition tools can acquire an entire snapshot of volatile memory; however, these typically require at a minimum the use of specific kernel versions as well as root privilege on the device. That is, the device needs to be rooted, forgoing essential security measures that are by default in place [47]. If the device's Android kernel does not allow inserting kernel modules (as is the case by default), creating a custom kernel may also be necessary. Other options involve hacking the device's firmware upgrade protocol, which requires the reverse engineering and availability of bootloaders [37], [48].

In this research, we focus on an incident scenario in which the victim is the device owner and will continue using the device beyond the scope of the investigation. Therefore, collection methods requiring irreversible modifications to the system are not viable. To this end, we instead aim to leverage the functionality provided by ART, which requires making customisations to the apps that can potentially be misused in an attack. While it is well within the remit of app developers to make apps forensically sound by dumping critical objects in memory in real-time, it is not

[5]https://developer.android.com/reference/android/os/Debug#dumpHprofData()

[6]https://android.googlesource.com/platform/art/+/6f4ffe4/runtime/gc/heap.cc#1086

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

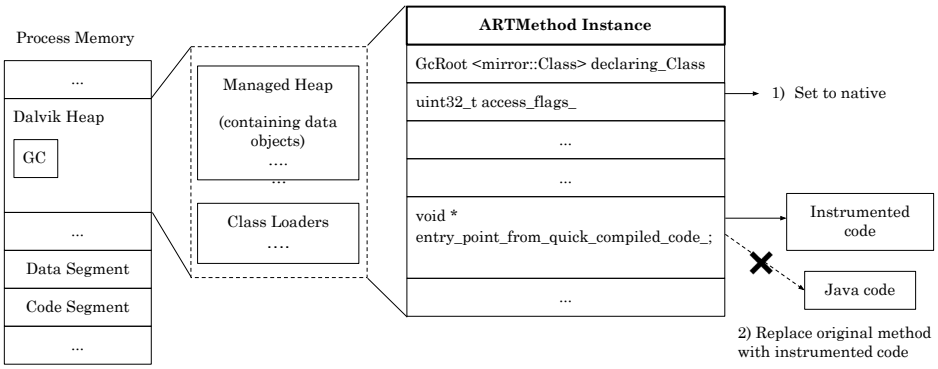


FIGURE 4: Instrumenting code via `ArtMethod` entry point.

always possible to know beforehand the objects required in a particular investigation, and taking a general approach to dumping objects that *might* be necessary may harm the performance of the app both in terms of storage and usability. JIT-MF takes a post-deployment app customisation approach by instrumenting apps to forensically enhance them so that they may timely dump the data objects that are critical to attack steps in a specific scenario. While this approach still faces several challenges, including code integrity checks that hinder app-repackaging, solutions exist [49] that make it possible to instrument an app *without* modifying its bytecode, bypassing standard code integrity checks.

E. ANDROID APP INSTRUMENTATION

Android apps are compiled and packaged as a single file known as an APK (Android Package Kit) with `.apk` as the file extension. The typical makeup of an APK consists of: an Android Manifest file providing essential information about the app, a `lib` directory, and additional directories for resources such as images/files required by the app. An app can make use of both Dalvik bytecode and Native code, which resides in the `lib` directory of the app, both of which can be decompiled to enable reverse engineering and modification of the code.

Customising apps post-deployment requires binary instrumentation techniques that modify the apps' behaviour during runtime. Binary instrumentation can be both static and dynamic. However, features in programming languages like reflection, challenges in comprehending compiled obfuscated bytecode, and dynamic code loading make a dynamic approach necessary in our case. While static binary instrumentation mainly requires modification of the compiled bytecode, dynamic binary instrumentation (DBI) involves two main components: i) Library injection and ii) Execution interception.

Library injection 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*, by patching either the app's Executable and Linkable Format (ELF) or it's Dalvik bytecode (`classes.dex`), or *dynamically* using system calls provided by the kernel such as `ptrace`[7]. Since the latter requires root privileges on the device, which is not granted by default and presents additional challenges, static library injection becomes the only choice. Static library injection involves decompiling the app and patching its contents to include instructions that load the library containing instrumentation code. The library itself is added to the app's `lib` directory. The app would then need to be repackaged and re-signed. While several tools exist to facilitate the process (e.g. `apktool`, `jarsigner`), anti-repackaging techniques may be employed to verify the integrity of the original app's codebase. Such anti-repackaging techniques involve code integrity checks that operate both at the device level and at the app level that check for the presence of app repackaging techniques.

Execution interception, the second component to DBI, allows control to re-direct towards instrumentation code and eventually back to the original execution. This can be achieved through hooking. *Inline hooking* is a form of hooking that requires inserting jump instructions in a code segment to direct the program execution process to the functional code required by the user. Most importantly, inline hooking allows *any* function/method (including internal functions) within the app to be hooked/intercepted. In the context of Android app instrumentation, since an app can run both native code and Dalvik bytecode, the means by which inline hooking is achieved must cater for both. Native code in the app runs directly on the CPU and has access to the Android framework using Java Native Interface, enabling 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.

Xposed-style method hooking [50], is an inline hooking approach used in Android, which makes it possible to replace the entire method body or introduce new code before

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

and after the original Java method invocation within an app. When a method is invoked, the execution goes through the injected code and then to the original code. This is done leveraging the runtime provided by the Android framework, ART. 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. Figure 4 shows the `ArtMethod` data structure which contains information about a particular Java method (method descriptors), such as 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_**). An ArtMethod instance can have either of four entry points *EntryPoint from Interpreter*, *EntryPoint from QuickCompiledCode*, *EntryPoint from JNI* and *EntryPoint from PortableCompiledCode* [51], depending on its modifier. When a method is executed, the instructions at the address of the entry point are executed. Instrumentation frameworks utilising Xposed-style method hooking, keep a copy of the values of the method modifier and entry point, and patch them to so that the new method modifier is `native` [8] (Step 1 in Figure 4) and the new entry point value (Step 2) of methods defined in the `ArtMethod` contains the address of methods containing instrumentation code. Once the instrumented code is executed, the original method modifier and entry point values are reset, and program execution continues. 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 to the hooked function containing instrumentation code [52].

Frida [53] is a dynamic instrumentation toolkit that can be used on Android devices and adopts inline hooking. Its embedded mode, known as Frida Gadget[9], implements both Xposed-style method hooking as well as native inline hooking to enable the instrumentation of an app's methods in a setting that does not require device rooting. Therefore, any function invoked from native or bytecode by the app at runtime can be intercepted by making customisations to the app without requiring irreversible changes to the device. Library injection is done statically by decompiling the app and placing the Frida Gadget shared library (*frida-gadget.so*) in the `lib` directory. The Frida Gadget library needs to be loaded inside the process memory at runtime as a second step. While bytecode is not easily read or manipulated, an intermediate language Smali, can be used to patch the code. The Frida Gadget library needs to be loaded at an early stage 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("frida-gadget")*'. This allows the Frida Gadget library and instrumentation code to be loaded in the process's memory as soon as the app is launched. Since patching Smali code involves making modifications to the app's codebase, any code integrity checks within the app that leverage the app's codebase, e.g. hash value of the bytecode (`classes.dex`), will be unsuccessful, resulting in the app not launching. Another approach to static library injection involves using LIEF [49], an ELF-centric method that patches the app's ELF to add the Frida Gadget as a dependency of native libraries embedded in the APK. This approach bypasses code integrity checks leveraging the contents of the `classes.dex` since the bytecode is not modified and the Frida Gadget library is loaded as soon as other native libraries are loaded in the app. This method, however, operates under the assumption that the app itself requires other native libraries to be loaded and that these libraries are loaded early on in the app's lifecycle. Following library injection, the app contents are then repackaged and re-signed to form an instrumented version of the same app that can be reinstalled. Finally, the Frida API includes the *Java.choose()* function, which enumerates live instances of the Java classes in memory, through the exported `Heap::GetInstances` function in `libart.so`, enabling the online collection and carving of evidence objects from memory.

III. RELATED WORK

In this work, we aim to provide a practical solution for rendering the collection of incident-related forensic artefacts as comprehensive as possible. Otherwise, no manual or automated process would be able to reconstruct the incident. We focus on stealthy Android attack vectors since these present an ongoing threat, with multiple recent incidents gaining a worldwide reach [9]–[11]. Crucially for incident response, the resulting reduced forensic footprint for recent stealthy attacks has also been demonstrated [13]. Zygote and binder infection combined with a rooting exploit [6], as well as app-level virtualisation frameworks [7] and third-party library infections [8] provide further attack vectors, resulting in similarly stealthy attacks which require a solution in terms of incident response, due to likely late detection and minimal (if at all) forensic footprint stored on the device.

The topic of Android messaging timelining has already received attention from various works [32], [54]–[59]. These works, however, mainly utilise disk images to retrieve valuable evidence. While stored data can be helpful, it is at the discretion of app developers which metadata is stored, and this is generally dependent on app functionality rather than the forensic hardening of the app. This is especially true in the case of stealthy attacks, whereby

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

[9]https://frida.re/docs/gadget/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

attackers purposefully cover all possible tracks, leaving no forensic footprint behind. Therefore, metadata critical to an investigative scenario may not be available at all from disk, and in cases where it is, this may not be true across different versions. Furthermore, with the increase in popularity of cloud-based messaging apps, less data becomes available locally to retrieve [60].

With stored data falling short of providing crucial forensic artefacts, especially in the case of stealthy attacks, recent works turn to memory as an additional forensic source of evidence [15]. Popular memory forensics tools and frameworks, like Volatility [46], [61] focus primarily on analysing content in kernel space, which require an entire memory image of the device. While tools like LiME [47] make it possible to retrieve such a memory image, the process of using such tools on Android devices is not straightforward. Root privileges are required on the device, and whereas this is typically easily achieved on systems via password access, mobile devices are by default unrooted. This means that rightful owners do not have root access as part of the safeguards in place. Rooting a mobile device is an irreversible action that removes these safeguards. In the context where the device owner is a victim of a cyberattack who will continue using the device after a possible incident, rather than a perpetrator whose device was confiscated, rooting the device is not a viable option. In the case of devices that have already been rooted, LiME still requires an additional kernel module for collection to be inserted in the kernel, which typically involves recompiling a custom kernel.

Tools that focus on kernel analysis, like those in the Volatility framework, have been effective in recovering essential artefacts such as running processes, network sockets etc. However, these fall short when the objective is to recover process-specific objects from memory that are critical to an investigation. In this case, such analysis is restricted to retrieving character strings, with no further context being available regarding the nature of the object within which these strings exist. Thus more recent memory forensics research has explicitly targeted *process memory forensics*, aiming to recover ephemeral data objects in memory crucial to forensic investigations [18], [20], [62]–[65].

Saltaformaggio et al. [62]–[64] created several tools whose aim is to reconstruct different objects types from residual artefacts in an Android memory image. VCR (Visual Content Recognition) [63] leverages static memory images of Android device cameras to recover photographic images. GUITAR [62] is a tool that rebuilds and redraws apps' UI screens from smartphone memory images based on the low-level definition of the Android GUI framework. The authors further extend this work with a more advanced memory forensic technique that performs spatial-temporal recreation of screens of Android apps from memory images [64]. While also targeting process memory to carve and parse objects of interest in memory, these tools are after app-agnostic objects from memory, as opposed to JIT-MF tools which aim to recover specific app objects from memory. Furthermore, the aforementioned tools focus on the collection of objects from a given memory dump. In contrast, within the JIT-MF framework, JIT-MF drivers are also responsible for the actual timely dumping of memory sections/objects that are of interest.

Droidscraper [20] is an Android runtime-based recovery technique designed for Android ART, that given an app's memory image, can reconstruct objects of types: Primitives, Arrays, Strings, and Complex classes. In this case, the recovery and reconstruction effort is designed based on a specific Android memory allocation scheme, as implemented by the Android Garbage Collector. Specifically, Droidscraper uses low-level data structure definitions as well as generic class and references constructs provided by the Android runtime library (`libart.so`) to provide investigators and malware analysts access to well-structured and forensically interesting objects found in an app's process memory. Similar to the works of Saltaformaggio et al. [62]–[64], Droidscraper takes a post-incident approach to recovering interesting objects from memory, and therefore its workflow starts with the acquisition of a process memory dump. In contrast, JIT-MF tools are intended for scenarios involving stealthy attacks and consequential late detection. Therefore timely memory dumps are required to ensure that the evidence critical to an attack scenario is present. While JIT-MF enables timely dumping of forensic artefacts from memory, unlike Droidscraper, the objects it is able to recover and reconstruct are app-specific and requires efforts in terms of reverse engineering, per app, as well as mitigating anti-repackaging techniques that might be in place.

DroidKex [65], much like JIT-MF, takes a dynamic approach towards collecting objects of interest from memory. It uses Frida (as is the case with MobFor) to timely collect memory dumps, specifically when `send` and `receive` functions are invoked during the app's runtime. Similar to the concept of trigger points in the JIT-MF framework, DroidKex uses the invocation of these native system calls as an indicator of an event. Whereas within the JIT-MF framework suspicious events can be defined in JIT-MF drivers, enabling the framework to cater for multiple attack scenarioS, DroidKex focuses solely on initialisation of TLS connections and the collection of TLS objects, which enables DroidKex to decrypt ongoing TLS traffic.

Similar to security monitors like REAPER [66] and MOSES [67], JIT-MF uses trigger points which, rather than being indicators for malicious events, such as permission misuse, are indicators of benign events that may be misused by an attacker. In contrast to typical monitors, JIT-MF dumps necessary memory contents for post-analysis at runtime, which is less costly than online analysis.

Having a custom specification (JIT-MF driver) underpinning a generic framework is common in digital forensics tools. Frameworks such as Autopsy and Volatility allow the addition of modules and plugins, which enable them to cater to a broad range of investigation scenarios. The concept can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

be even applied to reconstructing timelines from specific log files using custom analysers [40].

IV. JUST-IN-TIME MEMORY FORENSICS (JIT-MF)

The JIT-MF framework leverages static and dynamic app instrumentation to enable live process memory forensics by timely dumping key forensic artefacts (evidence/data objects) from memory that could constitute attack steps. JIT-MF's underpinning principles that distinguish it from state-of-the-art memory forensics tools are i) Real-time collection of critical evidence objects in volatile memory related to the key attack steps from target victim apps; and, ii) The timely dumping of specific fragments of process memory as specified by trigger points. Notably, in contrast to malware detection and forensics tools, JIT-MF tools focus on collecting evidence from misused benign target apps (rather than malware). Evidence data objects and trigger points are specific to investigation scenario/target app pairs, as defined within JIT-MF drivers. While we present JIT-MF in a messaging hijack setting, the proposed concept extends to the general case of attacks heavily leveraging benign apps.

A. TRIGGER POINTS

A typical memory dump contains those objects created and used by an app (both specific to the app and those belonging to the Android API) at the point in time when this is performed. Not all of these objects are relevant to the critical attack steps. For instance, in the case of a messaging hijack attack, we are only after the in-memory objects supporting the execution of messaging functionality and which may be hijacked during eventual attacks, e.g. the objects describing message contents and any additional ones that provide further context.

Trigger Point selection heuristic. The selection of trigger points is specific to two aspects: i) The attack scenario for which we want evidence to be collected, and ii) The set of operations the app executes in such scenarios. Optimal trigger point selection requires full knowledge of the specific app being analysed (and its version at the time). Given that most of the apps being analysed are expected to be third-party, assuming comprehensive knowledge of the app's codebase is not practical. Instead, trigger points are selected based on the following heuristic: Each app may handle operations related to key steps differently, however, the critical data objects involved in the operation *must* be handled in either of the following ways:

(i) **Stored** and **loaded** from storage;
(ii) **Transferred** over the network (e.g. Wi-Fi, 4G, etc.); or else
(iii) **Transformed** in some way (e.g. display on screen etc.).

In the case of a messaging hijack, key evidence objects comprise precisely those that contain the messages themselves (as defined by an app-specific class). In contrast, the operations related to these objects involve storing/loading messages from local content repositories and sending/receiving messages over communication networks. The heuristic provides the basis for trigger point selection.

Trigger Point categories. The selection of trigger points is not prescriptive. Therefore, any operation handling key data objects in the ways mentioned above may constitute a valid trigger point. However, the selected trigger points may adversely affect the runtime performance and stability of the concerned app, depending on their specificity. For instance, when receiving a new instant message, one can safely assume that the source code in the app handling the data object of interest (the evidence) must have made use of underlying network functionality at some point. Otherwise, the message would not have been received. In this case, generic network-related operations - such as the `recv` system calls - are considered viable, generic trigger points requiring minimal app reverse engineering effort since they can be derived without detailed knowledge of the app's codebase. However, such trigger points may not be as accurate as those selected with a more in-depth understanding of app functionality. The latter kind of trigger points encompasses app-specific methods, reflecting the particular invocation of the sought-after functionality, e.g. displaying the message in an app-specific GUI grid on the device screen. Such trigger points are expected to be more accurate, both in terms of producing timely memory dumps and in not being triggered too frequently (over-execution). That said, there may be instances in which generic trigger points can have trigger predicates or *filters* associated with them that decrease their invocation.

Overall, the varying degree of specificity of a trigger point reflects the amount of effort put into comprehending the codebase of an app. Therefore we categorise trigger points as follows, starting from the least specific (and requiring least reverse engineering effort) to the most specific, as described in Table 1. The first three categories are considered black-box, meaning they require the least knowledge of an app's codebase. The final category is considered white-box due to the need of having to peek inside an app's codebase for their identification.

B. OFFLINE VS ONLINE EVIDENCE COLLECTION METHODS

Once triggered, memory dumps can comprise entire ART heap sections as in `hprof` dumps, with subsequent evidence collection happening offline using an `hprof` parser, e.g. Eclipse MAT. A more frugal approach leverages ART's Garbage Collector (GC) to dump solely the key evidence objects in memory at runtime. In this setting, evidence objects are collected during the dumping process itself in an online fashion. Both approaches are compatible with non-rooted devices. We also distinguish between the object collection *carving* and *parsing* stages. By object carving, we refer to the process of identifying and extracting in-memory objects either from live memory (in the online case)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 1: Trigger point categories.

Trigger Point Category	Classification	Description
Native Runtime (RT)	Black-box	Generic native runtime system calls
Device Events	Black-box	Generic events related to the device state
Android & 3rd party library APIs	Black-box	Android API calls
App specific APIs	White-box	API calls specific to the app

or from a partial memory image (in the offline case). On the other hand, object parsing deals with extracting meaningful information out of 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 reverse engineering. In the case of offline evidence collection, the object carving and parsing stages are carried out offline; that is, at a later stage on a partial memory image that was dumped. 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 either online or offline. That is, objects may be parsed either in real-time or at a later stage (offline), depending on the computational effort required to parse the object and retrieve the meaningful information.

C. JIT-MF DRIVERS

While JIT-MF defines those common steps followed by every JIT-MF tool, those aspects specific to the investigation scenario/target app pair at hand are described and eventually implemented by JIT-MF drivers.

Figure 5 illustrates the involvement of these drivers in the JIT-MF framework. The generation of a JIT-MF driver requires, as input, knowledge of both Android System Services and the investigation scenario/target app pair at hand. These equip JIT-MF driver developers with the necessary insight to select the appropriate trigger points and evidence objects for an investigation scenario/target app pair. Conceptually, JIT-MF driver definition starts with identifying the in-memory evidence data objects of interest, which may correspond in some way to attack steps. Next comes trigger point selection, which corresponds to function hooks being placed in native or managed code at the implementation level. Hooks placed in native code are done through native inline hooking, whereas those placed in managed code are done through inline hooking of the `ArtMethod`. Trigger points can be rendered more specific through complementary conditions defined over function arguments (*trigger predicates*). The collection method must be defined as either online or offline, specifying whether object carving is carried out before memory dumping or else afterward. In the online case, an associated flag indicates whether the corresponding object parsing is carried out on live memory or later on the dumped raw object bytes. Finally, a sampling method is defined to maintain a manageable amount of dumps, especially when a generic trigger point is selected that is invoked many times. Listing 1 presents a template generically describing JIT-MF drivers.

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 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[10]. Lines 6-12 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 with $\{val1|val2|...\}$. Function parameters are identified by the final parenthesis $()$, and these correspond to internal functions in the drivers (lines 21-54). $carve\_object\_type_j\_offline$ and $parse\_object\_type_j\_offline$ (lines 34-35 and 44-45 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 in the case of Listing 11, a variable `timestamp` is set by one function and used by another, and is made accessible via the $Globals$ variable.

$init()$ presents the only interfaces exposed to the JIT-MF tool's main environment. It is called during tool initialisation and sets up the event $Triggers$ 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 $Triggers$. Triggers 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 (see Listing 3). $carve\_object\_type()$ 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

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

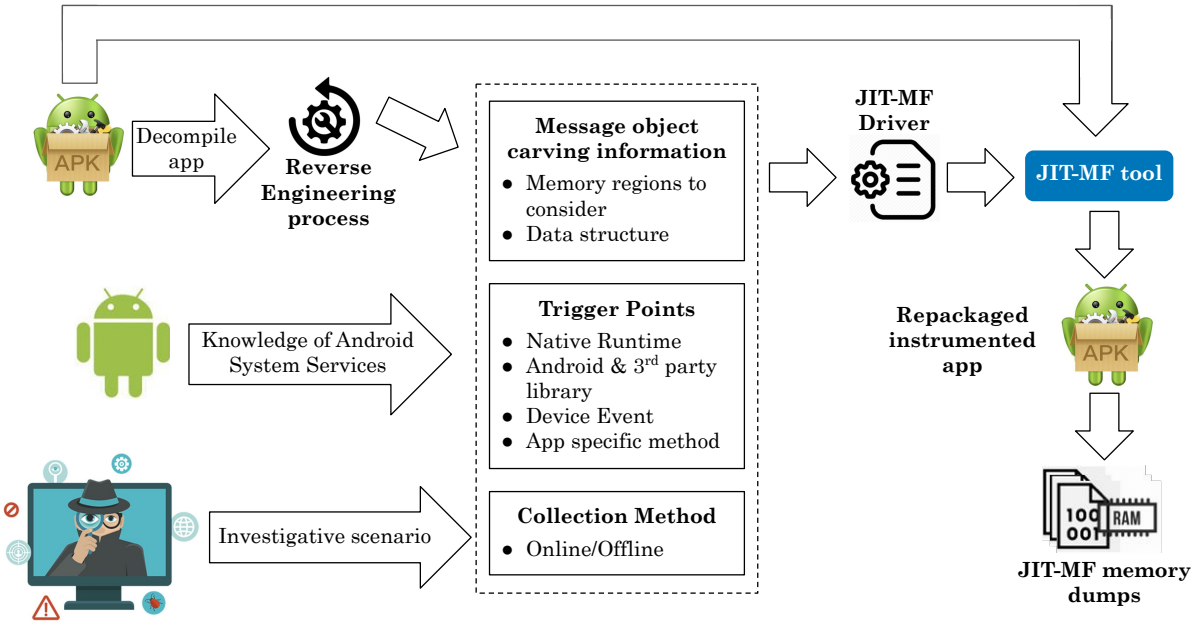


FIGURE 5: JIT-MF drivers.

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 2 presents a specification for the runtime which can be assumed by JIT-MF drivers 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. Lines 3-11 are process memory interacting functions, starting off $list\_memory\_segments()$ to make sure 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 $set\_memory\_permissions()$, as well as checking memory content through $read\_memory()$. While for offline $Collection$, calling $dump\_memory\_segment$ suffices, for online collection the driver is also required to carve objects and parse their fields. $dump\_native\_object()$ and $dump\_rt\_object$ are utility functions that do just that, by first locating the $Evidence\_object$ in memory, then executing 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 being used to manage native objects. $call\_native\_function()$ and $call\_rt\_function()$ functions are utility functions that may be needed by both driver and runtime functions. Finally, $append\_log()$ (line 11) is responsible to produce the actual JIT-MF dump to the location specified by the driver's $Log\_location$.

V. DEMONSTRATING THE VALUE OF JIT-MF

To answer ***RQ1*** and further motivate ***RQ2*** and ***RQ3***, we first demonstrate the value that JIT-MF adds to incident response by dumping in-memory forensic artefacts of otherwise elusive evidence, resulting in the generation of more comprehensive forensic timelines. Specifically, we answer the question: ***"Does evidence of attack steps recovered through JIT-MF's real-time triggering of memory dumps, enrich the corresponding timelines generated?"***

Below we define how the JIT-MF drivers were developed for the case studies considered, the methodology used to merge JIT-MF output as a forensic source with all other available sources — producing the resulting timelines of the corresponding incidents — and experimentation results.

A. JIT-MF DRIVER DEFINITION

A minimal effort approach is taken when generating JIT-MF drivers for the case studies considered, since the aim of *RQ1* is to validate JIT-MF as a technique, by demonstrating how it can enrich forensic timelines generated by existing mobile forensics tools. Given the trigger point categories defined in Table 1, of which the black-box categories require the least code comprehension effort, the $Triggers$ selected for the drivers should fall in this category. On the other hand, $Evidence\_objects$ identification and any corresponding object parsing required should follow from white-box

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Listing 1: JIT-MF driver template.

```
Driver_ID: string
Scope: <app, incident_scenario>
Processes:[optional <namespace.classname>,...]

/* Attributes */
Evidence_objects: {<event: string, object_name: string, carve_object_type(), parse_object_type(), {
    trigger_ids}>, ...}
Collection_method: {online | offline}
Parsing_method: {online | offline}
Triggers: {<trigger_id: string,<hooked_function_name: string, level: {native|rt}, trigger_predicate
    (), trigger_callback()>>,....}
Sampling_method: sampling_predicate()
Log_location: string
Globals: optional {<key,value>, ...}

/* Exposed interface */
bool init (config: {<key,value>, ...}) {
  for entry in Triggers:
    place_native_hook() ⊕ place_rt_hook();
}

/* Internal functions */
bool trigger_predicate_i(params: {<key,value>, ...}) {
  decide on whether to fire the corresponding trigger;
}
void trigger_callback_i(thread_context: {<key,value>, ...}) {
  if trigger_predicate_i() && sampling_predicate():
    perform memory forensics on the current app state;
}
[object: address,...] carve_object_type_j(from: address, to: address) {
  if Collection_method == online:
    attempt object carving in the given memory range;
  else:
    carve_object_type_j_offline(from, to);
}
@OFFLINE
[object: address,...] carve_object_type_j_offline(from: address, to: address) {
  use an hprof parser to carve object_j in the given memory range;
}
[<field:value>,...] parse_object_type_j(at: address) {
  if Parsing_method == online:
    parse object fields starting at the given address;
  else:
    parse_object_type_j_offline(at);
}
@OFFLINE
[<field:value>,...] parse_object_type_j_offline(at: offset) {
  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: {<key,value>, ...}) {
  decide on whether to follow up a trigger by a memory dump;
}
```

analysis of the apps concerned, meaning that some form of minimal app code comprehension is still needed. The finalized methodology adopted follows:

- *Evidence_objects*: These objects are identified as those whose presence in memory, in the context of a specific trigger point, implies the execution of some specific app functionality, possibly a delegated attack step. Not all objects are associated with the same level of granularity concerning app events; some objects may be highly indicative of a detailed app event, e.g. a message object with an attribute *sent=true|false*, others may only reflect vague app usage across a time period. Therefore when selecting evidence objects, one has to keep in mind how tightly coupled the presence of the objects is with the app functionality that needs to be uncovered.
- *Triggers*: Taking into account an attack scenario, corresponding target app functionality, and the associated

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Listing 2: JIT-MF driver runtime.

```
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);
```

evidence objects, trigger points are selected based on the code that processes the said objects, specifically concerning data object processing as defined in Section IV-A. As yet, we are limited to introducing trigger points only in *one* of the app's processes.

B. FORENSIC TIMELINE ANALYSIS

Forensic timeline generation considers all those 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. We opt for local device collection, rather than cloud or backups, to facilitate experimentation whenever the same data could be obtained from multiple sources. These forensic sources, as explained in Section II-B, are representative of those targeted by state-of-the-art mobile forensics tools, typically also requiring device rooting or a combination of hardware-based physical collection and content decryption. These baseline sources are complemented by those obtained through the use of JIT-MF drivers.

Figure 6 shows the processes that transform the evidence obtained from the aforementioned forensic sources to finished forensic timelines. This pipeline is based on Chabot et al.'s [68] methodology. It revolves around the creation of a knowledge representation model as derived from multiple forensic sources and 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, which are the raw forensic artefacts collected from the 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 subsequently be correlated based on common subjects, objects, as well as temporal relations, or expert rule-sets. Event correlation starts with a seed event, which is deemed suspicious by investigators due to its unusual nature, or else pinpointed by the user as not being the result of intended device usage. Relations established by this process correspond either to a relation of composition or causation.

The first three steps in Figure 6 consist of forensic footprint extraction. For JIT-MF, we refer to a combined dump containing unique, readily carved and parsed memory objects. All sources are decoded and merged as a Plaso [41] super timeline using the `psteal` utility, and for which we developed a JIT-MF Plaso parser that processes readily-parsed JIT-MF dumps into a single JSON file. A loader utility was developed for Step 4 that traverses the super timeline and populates the knowledge model, which we store in 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. We stick to a flat model for this experimentation, with events considered atomic and their associated subjects and objects corresponding to message recipients and content, respectively. 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, in step 6, we used `Timesketch` [42] for visualization.

C. EXPERIMENTATION

To demonstrate the added value JIT-MF tools provide during incident response by enriching resulting forensic timelines produced with other mobile forensic sources; we consider a suite of case studies inspired by real-life accessibility attack scenarios that target messaging apps. Each case study assumes a high-profile target victim ("John"), whose Android mobile device stores confidential data. John makes use of multiple messaging apps which 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

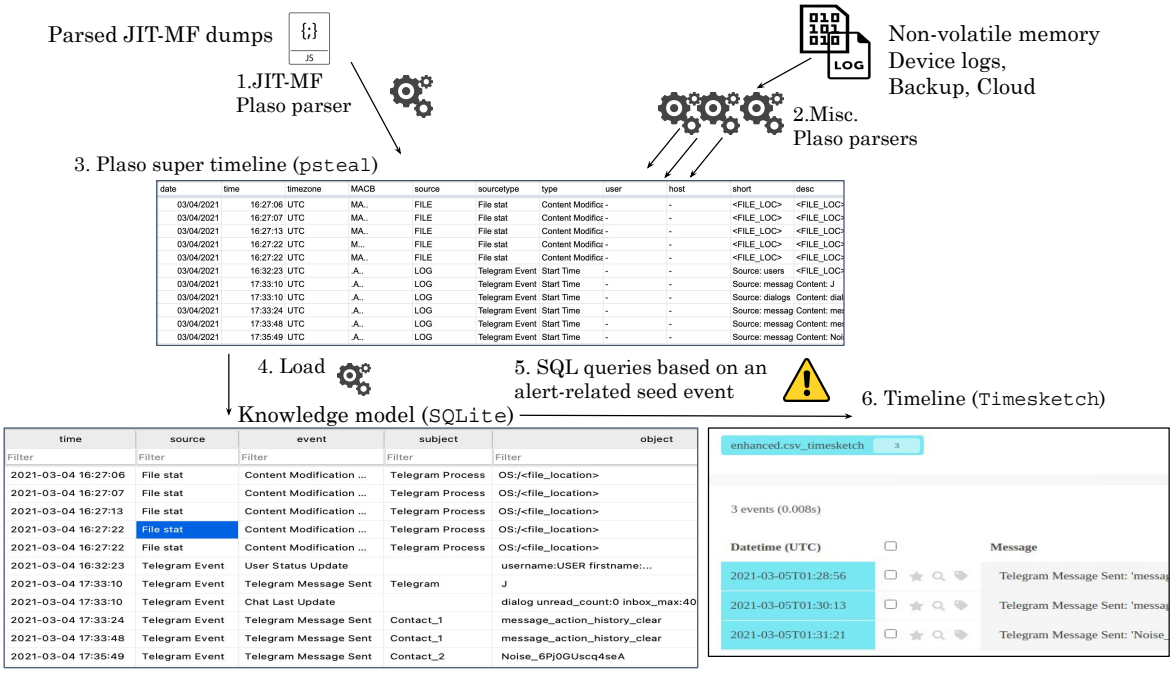


FIGURE 6: The forensic timeline generation processes.

victim of a long-term stealthy attack.

Setup. Pushbullet(v18.4.0), Telegram (v5.12.0), and Signal (v5.4.12) are popular SMS and Instant Messaging apps, respectively, used in our case studies. The messaging hijack scenarios considered involve spying and crime-proxying. 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. Since most state-of-art forensic sources require a rooted device, and rooting a device is a non-reversible action, we opt for a rooted emulator in our experiment, which also enabled ease of automation. The emulator used was Google Pixel 3XL developer phone running Android 10. The JIT-MF driver runtime was provided by a subset of the Frida[11] runtime, and JIT-MF drivers were implemented as Javascript code for Frida's Gadget shared library. Listings 3 - 8 outline the JIT-MF drivers created and used in the case studies considered in this experiment. Specifically, all the drivers have the following attributes: $Collection\_method = online$, $Sampling = 1\ in\ 5$ (active for a second every 5 seconds) and $Log\_location = / < external\_storage > /jitmflogs$. $Parsing\_method = online$ except in the case of Pushbullet drivers where the object requires the use of regular expressions and hence more computational effort, therefore is left for post-processing (offline). Table 2 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. Apart from the JIT-MF Plaso parser[12], 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: the emulator is started, the target app is instrumented, legitimate traffic (noise) and malicious events are simulated within a controlled setup, forensic sources of evidence are collected (both baseline and JIT-MF sources), and timelines are produced based on a knowledge model. The output of these steps are the following generated timelines:

- A *Ground Truth Timeline* generated by logging the individual attack steps of the executed accessibility 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 dumps.

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. In each case study, attacks comprise three malicious events. Noise is generated with respect to this value. Some of the attacks used in these case studies target victim apps that use rate-limited API calls to a server backend, which only allows 150 consecutive calls to be made from the same device.

[11]https://frida.re/docs/android/

[12]https://gitlab.com/mobfor/jitmf_experiments_resources/-/blob/main/Determining_JIT-MF_Value_Experiment/git_repos/plaso/plaso/parsers/jitmf.py

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Since each attack comprises three such events per case study, and the API call limit is 150, each case study is simulated fifty times — each time obtaining the timelines above.

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 with respect to 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 app in use. Therefore, the matching criteria for a matched event between the generated timelines and the ground truth timeline are adjusted in the case studies to benefit from the evidence typically found in baseline forensic sources.

1) 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 going by the username "Alice" on Telegram. The attacker misuses the victim's Telegram app to send messages to "Alice" and instantly deletes them.

Setup. John makes use of 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 decides to use 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 within the attack 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 3.

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 7 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.

2) 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 going by 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 4.

Similar to the previous case study, the evidence object is the Signal message itself. Signal does not make use of the `send` system call however when sending a message. The `write` system call is used as a trigger point, which writes to the local database and over the network.

Seed Event: Subject: Alice, Object: *specific keywords*, Event type: Message Sent, Time: last 7 days

Matching criteria: An event stating that a message was sent to Alice's number.

3) C: Pushbullet Crime-Proxy

Accessibility attack. John's Facebook credentials are stolen by an attacker using a phishing accessibility attack akin to Eventbot [9]. 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 prior to entering a meeting. The attacker notices the decrease in activity and decides to use 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 at the end of the month from his telephony provider, attributing most of the cost to message sending. He notices a new number that is not on his contact list and takes his phone to be examined. His phone is already equipped with a JIT-MF driver as shown in Listing 5.

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.

4) D: Telegram Spying

Accessibility attack. An accessibility attack targets John's Telegram app and is used by an attacker to intercept mes-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 2: 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	JIT-MF memory dumps	`adb pull`	No	JIT-MF parser

sages 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 makes use of his Telegram app regularly 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 6.

In the case of spying, one of the attack steps involves the reading of 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. That is, 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 any time after the sending time to avoid including data related to when the message was initially sent or received.

5) 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 is already equipped with a JIT-MF driver as shown in Listing 7.

Similar to the previous case study, the evidence object is the Signal message that was intercepted. Signal is not a cloud-based app and uses solely on-device storage. Therefore we select the `open` system call which is used to open the database file from which messages are loaded to be 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.

6) 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 makes use of his SMS app regularly 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 is immediately intercepting all of John's ongoing SMS activity.

Investigation. John's phone is already equipped with a JIT-MF driver as shown in Listing 8.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

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 automatically have been intercepted. 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.

Listing 3: JIT-MF driver for Case Study A: Telegram Crime-Proxy.

```
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 4: JIT-MF driver for Case Study B: Signal 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;
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access


```
}

/* Helper function */
datetime get_time(){
  return current time;
}
```

Listing 5: JIT-MF driver for Case Study C: Pushbullet Crime-Proxy.

```
Driver_ID: PUSHBULLET_CP
Scope: <pushbullet, crime-proxy>
Processes:[com.pushbullet.android.sms.SendSmsReceiver]

/* 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 6: JIT-MF driver for Case Study D: Telegram Spying.

```
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
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access


```
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 7: JIT-MF driver for Case Study E: Signal Spying.

```
Driver_ID: SIGNAL_SP
Scope: <signal, spying>
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access


```
/* 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 8: JIT-MF driver for Case Study F: Pushbullet Spying.

```
Driver_ID: PUSHBULLET_SP
Scope: <pushbullet, spying>
Processes:[com.pushbullet.android.sms.SendSmsReceiver]

/* Attributes */
Evidence_objects: {<"Pushbullet Message Synced","org.json.JSONObject", carve_json_object(), parse_json_object(), {"1"}>}
Collection_method: online
Parsing_method: offline
Triggers: {<"1",<"read",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='{"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;
}
```

D. RESULTS

Table 3 shows a comparison between 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); 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. Where the value is '-', no events were captured. 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 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.

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 finer-grained evidence (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. Due to the finer-grained metadata available in JIT-MF forensic sources, we can say that even in the scenarios where JIT-MF timelines

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 3: Forensic timeline comparisons.

Case Study	Seed event - Correlation	Baseline			JIT-MF Timeline		
		Recall	Precision	Timeline difference (Jaccard dissimilarity) in attack events	Recall	Precision	Timeline difference (Jaccard dissimilarity) in attack events
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

are equivalent to the baseline in event sequences, these can provide the investigator with richer timelines through more informative events.

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

Primary contributors to timeline dissimilarity. JIT-MF timelines were most dissimilar from the ground truth in the last case study. The main differences from the other case studies here were: i) Many of the app's functionality was delegated to a sub-process that was not instrumented, and ii) The evidence object defined in the JIT-MF driver was coarser-grained (a JSON object containing "push" event that synced changes). 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.

To mitigate this, modifications needed to be made to the JIT-MF driver template to cater to apps and incident scenario pairs, whose trigger point is executed in another subprocess of the app (not the main process). Listing 1 line 3 shows how JIT-MF drivers can be fine-tuned, so that specific app processes are instrumented rather than automatically instrumenting the main app process. This allows JIT-MF to be more effective across the different case studies considered.

Furthermore, while JIT-MF timelines are more similar to the ground truth timeline than baseline timelines in cases involving spying (D-F), 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 defined in the JIT-MF driver is the fine-grained message object that contains metadata tightly linked to the event itself. In spying scenarios, we are after coarser-grained events (an indication of a chat being intercepted/synced) since key objects in memory are either absent or do not contain indicative metadata of the ongoing event.

JIT-MF timeline sequence accuracy. When performing order-sensitive comparisons using *Kendall Tau coefficient*, we were able to 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 performance of the app, 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[13], 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,

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

respectively, with JIT-MF drivers enabled; the performance penalty overall was less than 3.53%.

While Janky frames are an indicator for how smooth the UI of the app is while it is running, this does not factor in performance issues such as the app crashing, a few instances of which we observed during experimentation. Furthermore, while the experiment ran for a relatively short period of time, logs generated by JIT-MF and stored on the device, continue to accumulate over time until they are collected and removed from the device by a potential investigator. Therefore, if the magnitude of these logs increases exponentially over time, the device would be rendered unusable. Both of these issues contribute towards the feasibility of JIT-MF in a realistic setting and require further experimentation.

The results obtained address ***RQ1***, as we demonstrate that in an emulated setting, JIT-MF is successful in timely dumping data objects in memory related to evidence of attack steps. Furthermore, the evidence collected through the use of JIT-MF is not available in any other forensic source and is crucial to constructing a comprehensive, enriched forensic timeline.

VI. MOBFOR: A TOOL BASED ON JIT-MF

While shown to be of potential value in an emulated setting, by enriching forensic timelines with evidence recovered solely by JIT-MF sources (Section V), bridging the gap to real-world deployment requires further experimentation.

The results shown in the previous section outline some of the practical challenges that JIT-MF faces in terms of app stability. Furthermore, given that JIT-MF is intended to work in an incident response setting, enriching forensic timelines by complementing existing forensic sources, we enlist the following objectives that should be met by any JIT-MF tool:

1) A JIT-MF driver is needed for each pair of app and incident, therefore code comprehension and/or reverse engineering efforts must be kept at a minimum.
2) Effective sampling is required to minimize both the burden on storage requirements as well as the degradation of user experience (app stabilisation) due to additional processing.
3) The output should be in such a format that is easily ingested by analysis tools.
4) Address the challenges that arise with app instrumentation, namely: i) Multi-process apps ii) Obfuscation resistant apps, and iii) Code integrity checks.
5) A chain of custody must be kept to ensure that all the operations carried out on the device are well documented and any recovered evidence may be admissible in court [25].

With the above objectives in mind, we set out to answer the second research question: ***"Given a set of requirements for a JIT-MF-based tool, MobFor, what are the possible solutions that would render the tool practical?" (RQ2)***. To this end, we aim to implement *MobFor*; a JIT-MF tool that meets these objectives, enabling JIT-MF to function in a real-world setting. We reproduce prior experimentation that provides guidelines regarding trigger point selection [21], addressing the first objective, and conduct novel experimentation to observe the effects of sampling, addressing the second objective. Finally, we discuss MobFor implementation details addressing the final three objectives.

A. TRIGGER POINT SELECTION

While based on the JIT-MF driver template in Listing 1, the conceptualised implementation of the drivers created (Listings 3 - 8) shows that developers of JIT-MF drivers still have to perform the arduous task of finding valid trigger points and selecting evidence objects that can be considered critical evidence in a specific stealthy attack scenario. While the selection of evidence objects may be obvious (especially in the case of open source apps), since this constitutes the object in memory that needs to be dumped, the selection of trigger points may not be as straightforward and requires more effort in comprehending the codebase (reverse engineering) of an app to understand its functions. To this end, we perform a comparative analysis of the different trigger point categories defined in Table 1, based on their ability to timely dump critical evidence objects from memory and observe their associated storage overhead costs.

1) Experimentation

Four messaging hijack case studies were set up for experimentation purposes, encompassing SMS (via the closed-source Pushbullet app) and IM (via the open-source Telegram app): 1) SMS Crime-proxy, 2) SMS Spying, 3) IM Crime-proxy and 4) IM Spying.

All four attack scenarios were implemented as extensions to Metasploit's Meterpreter for Android[14]. For SMS-related attacks, the accessibility malware typically first sets up a Pushbullet installation and signs in using phished credentials. The full setup comprises: Pushbullet v17.7.19 and Telegram v6.1.1 Android apps instrumented with JIT-MF drivers through MobFor, both installed on an Android 10 emulator. The JIT-MF drivers used implemented both online and offline evidence collection methods as described in Section IV-B, leveraging Frida's `Java.choose()` and Android's API `Debug.dumpHprofData()` respectively. In the case of Pushbullet, we assume a legitimate user did the initial installation. To measure effectiveness, we search for the proxied/stolen messages in the resulting JIT-MF memory dumps output and note whether or not they were found. All attacks were repeated ten times since it sufficed to reach convergence for all measurements taken.

Eight JIT-MF drivers were created, each having a different trigger point (TP) per investigation scenario/target app. Two trigger points were selected for each category defined in Table 1, attempting to leverage all available candidate

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

trigger points in terms of disk input/output, network send/receive, and miscellaneous object transformations, following the heuristic described in Section IV-A, while keeping an eye on comparing black-box and white-box trigger point categories. The chosen TPs are listed in Table 4, where TP1 is either file/disk or object transformation-related, and TP2 is network-related. $Trigger\_predicate()$ were used for better specificity and device event trigger point checks are triggered based on their native category counterpart, so the instrumentation checks for increased directory size, after a `write()` call is made.

2) Results

Table 5 compares the trigger points based on accurately dumping evidence objects related to the proxied or intercepted SMS/IM messages over ten runs per attack. The first six rows are the results obtained for the black-box trigger points, while the next two are for the white-box. The results column-wise, i.e. across trigger point categories, show that the hypothesis that white-box trigger points are more accurate than their black-box 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 points could be possible solely within the black-box categories, which are those requiring minimal app-specific knowledge. Furthermore, the results presented in Table 5 also show that the effectiveness obtained by using offline and online collection methods have very similar results.

Black-box trigger point categories show promise. The fact that black-box trigger points can be as effective and efficient as white-box ones bodes well for the first objective set out for MobFor, i.e. requiring less code comprehension/reverse engineering efforts to implement JIT-MF drivers. While both effectiveness and runtime overheads so far do not overwhelmingly favour any of the three black-box categories across the board, it seems that certain trigger point categories might be less resource-intensive for some apps and more for others. This, however, merits further field observation once MobFor is deployed.

Considering only the best-performing trigger points within each black-box category, per investigation scenario (highlighted in grey), we notice that across the four scenarios *Device Event* trigger points were the least effective, with the other two categories at a tie. Furthermore, in the cases where the trigger points in this category 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 points in the other two black-box categories. These results, therefore, suggest that in the case of *Device Event* trigger points, several trigger points may have to be evaluated before deployment. While for certain app functionality we can assume underlying app events (e.g. `send` syscall in case of an outgoing message over the network), the effects that this functionality may have on the devices may differ depending on app usage as well as on 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 *Device Event* trigger point that monitors an increase in app directory size may not be entirely 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 to generate better *Device Event* trigger points targeted to a scope.

Furthermore, considering the two most successful categories within each black-box category (*Native RT* and *Android & 3^{rd} Party APIs* trigger points), results show that the distinction in effectiveness between file/disk-related trigger points (TP1) and network-related trigger points (TP2), remains unclear, as results vary substantially between the two types of trigger points even in the same category and scenario. That said, network-related fared consistently in the scenarios involving the instant messaging (IM) app, Telegram. The scenarios involving SMS, show mixed results, however, demonstrating that even within the context of message hijack attacks, different types of trigger points leveraging both file/disk and network might be required depending on the type of messaging app involved in the scenario.

Overhead storage costs. Whilst results show that black-box trigger points do not necessarily incur higher storage costs, with online collected dumps requiring as little as 0.1kB to be effective, 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, which in our case studies corresponds to SMS/IM sending/viewing, dumps are expected to be very frequent. While perhaps SMS is of less concern 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.

When opting for offline evidence collection, we note that while the average dump size required by online collection is around 143kB, that required by the offline method is 203MB (an order of magnitude more on average), per attack scenarios and trigger points chosen.

B. TRIGGER POINT SAMPLING

Any app which is forensically enhanced with JIT-MF drivers using some JIT-MF tool must remain usable by the device owner. The motivation for introducing sampling (second MobFor objective) arises from the need to minimise on-device storage cost, which is already present even in the initial representative study presented in the previous section. Furthermore, by addressing storage costs, we simultaneously

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 4: Trigger points selected.

Case Study	TP Category	TP #	Event selected	Trigger Predicate
(SMS) Pushbullet - Crime-proxy	Native Runtime	TP 1	write()	To file descriptor
		TP 2	read()	From TCP Socket as file descriptor
	Device Event	TP 1	Increase in app directory size	app_directory_path = Context().getFilesDir().getParent()
		TP 2	Increase in network traffic	Incoming
	Android & 3rd Party APIs	TP 1	android.content.ContentResolver.insert	-
		TP 2	android.telephony.SmsManager.sendTextMessage	-
	App specific APIs	TP 1	com.pushbullet.android.sms.SmsSyncService.a	-
		TP 2	com.pushbullet.android.providers.syncables.SyncablesProvider.insert	-
(SMS) Pushbullet - Spying	Native Runtime	TP 1	read()	From file descriptor
		TP 2	write()	To TCP Socket as file descriptor
	Device Event	TP 1	Increase in app directory size	app_directory_path = Context().getFilesDir().getParent()
		TP 2	Increase in network traffic	Outgoing
	Android & 3rd Party APIs	TP 1	android.content.ContentResolver.registerContentObserver	-
		TP 2	com.google.android.gms.gcm.GcmReceiver.onReceive	-
	App specific APIs	TP 1	com.pushbullet.android.sms.SmsSyncService.a	-
		TP 2	com.pushbullet.android.gcm.GcmService.a	-
(IM) Telegram - Crime-proxy	Native Runtime	TP 1	open()	File descriptor
		TP 2	send()	To TCP Socket as file descriptor
	Device Event	TP 1	Increase in app directory size	app_directory_path = Context().getFilesDir().getParent()
		TP 2	Increase in network traffic	Outgoing
	Android & 3rd Party APIs	TP 1	android.widget.EditText.setText	-
		TP 2	android.app.SharedPreferencesImpl\$EditorImpl.commitToMemory	-
	App specific APIs	TP 1	org.telegram.tgnet.ConnectionsManager.native_sendRequest	-
		TP 2	org.telegram.messenger.SendMessagesHelper.performSendMessageRequest	-
(IM) Telegram - Spying	Native Runtime	TP 1	open()	File descriptor
		TP 2	recv()	From TCP Socket as file descriptor
	Device Event	TP 1	Increase in app directory size	app_directory_path = Context().getFilesDir().getParent()
		TP 2	Increase in network traffic	Incoming
	Android & 3rd Party APIs	TP 1	android.view.ViewGroup.dispatchGetDisplayList	-
		TP 2	android.app.ContextImpl.sendBroadcast	-
	App specific APIs	TP 1	org.telegram.ui.Cells.DialogCell.update	-
		TP 2	org.telegram.messenger.MessagesStorage.putMessages	-

TABLE 5: Trigger point effectiveness: % accuracy over ten runs. Values highlighted in grey represent the trigger points that performed best within each black-box category, per investigation scenario.

Trigger point Classification & Category / Scenario			Crime-proxy - IM		Spying - IM		Crime-proxy - SMS		Spying - SMS	
			Online	Offline	Online	Offline	Online	Offline	Online	Offline
Black-box	Native RT	TP 1	100	80	100	100	30	80	80	80
		TP 2	50	50	100	100	100	100	0	30
	Device Event	TP 1	40	40	100	90	50	80	20	30
		TP 2	50	50	100	100	100	100	0	30
	Android & 3rd Party APIs	TP 1	90	90	100	100	100	100	0	0
		TP 2	80	60	100	100	100	100	80	80
White-box	App specific APIs	TP 1	100	100	100	100	0	0	0	0
		TP 2	0	0	60	60	100	100	80	80

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

aim to address any resulting app performance degradation due to saturation in resource usage that may cause the app to crash.

In the case of JIT-MF tools, storage costs can be attributed to two factors: i) The size of memory fragments dumped (either specific data object in the case of online collection or the entire memory dump size in the case of offline collection); and ii) The number of times a trigger point is invoked causing the trigger point callback to be executed and a memory dump to be taken (*TP Frequency*), which can be limited to an extent by declaring a $Trigger\_predicate()$.

While JIT-MF drivers can cater to both offline and online collection (see line 7 in Listing 1), depending on the need of a particular scenario, we choose to equip MobFor with JIT-MF drivers using online evidence collection. This decision follows the results obtained in Section VI-A2 , which show online collection to be the least costly of the two in terms of storage and just as effective as its offline counterpart (see Table 5). By taking an online approach, the size of the memory dumped upon each trigger point invocation is reduced significantly; however, the burden on the device storage resources can still accumulate to large quantities depending on the TP frequency.

To this end, we propose two sampling methods shown in Table 6, both of which aim to minimise the number of times a trigger point is invoked, thus reducing TP frequency and overall storage costs over time. We define *sampling window* as the range of possible values from which a sampling value is randomly chosen that determines whether or not 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 size of the sampling window 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 needs to gauge how many times a trigger point is invoked in an acceptable range of time (assuming typical app usage involving message sending and loading - see Section VI-B3). For instance, if a chosen trigger point is hit very frequently during the app's runtime when a single messaging event occurs, 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 critical objects 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 of time 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 of time 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 [18], [20] show that enhanced 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, when adopting sampling, while objects from memory are collected less frequently (or fewer memory dumps are taken - in the case of offline collection), this does not necessarily mean that fewer critical data objects are captured. Garbage collection algorithms running on the device should work in JIT-MF's favor, such that critical objects that were 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, we are optimistic that this decrease will be minimal.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 6: 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)

1) Experimentation

Setup. We use the JIT-MF drivers created in previous experiments (see Listings 3 - 8) and combine the drivers of a specific app into one robust driver that caters for message hijack attacks (both spying and crime-proxy). We equip MobFor, a JIT-MF tool, with these three JIT-MF drivers targeting messaging hijack attacks, misusing Pushbullet(v18.4.0), Telegram(v5.12.0), and Signal(v5.18.5). The drivers use black-box (Native RT) trigger points, and the evidence objects consist of the `Message` object unique to each app. The experiment is carried out on an Android 10 emulator for ease of automating message sending.

This experiment aims to observe how random sampling values selected from increasing sampling windows, for each sampling method proposed in Table 6, affect the app's performance at runtime (in terms of stabilisation; i.e. number of crashes and device storage resources) under typical app usage. For the scope of this experiment, we consider typical messaging functionality as 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, we conduct preliminary observations 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.

The `logcat` crash buffer is used to determine whether or not the app in question crashed during either of the runs. 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 were not outputted, we assume that the app has crashed. Finally, we determine the volume of the evidence produced by MobFor.

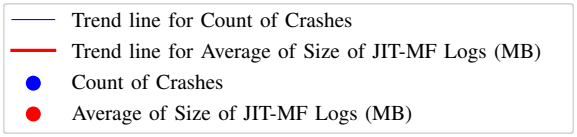
We carried out our experiment in the following manner; for each app, preliminary runs were used to determine the interval between the different sampling windows used, up to the maximum value in the sampling window range, per sampling method. For each sampling window, the JIT-MF driver is created with the relevant implementation of the *sampling_predicate*() function. Any residue logs are deleted, and Janky frame statistics are reset. The updated 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. It was noted that if an app hangs/crashes during the process described above, it never "recovers" unless it is closed and reopened. Therefore, the number of messages sent was kept to a minimum while reflecting practical usage.

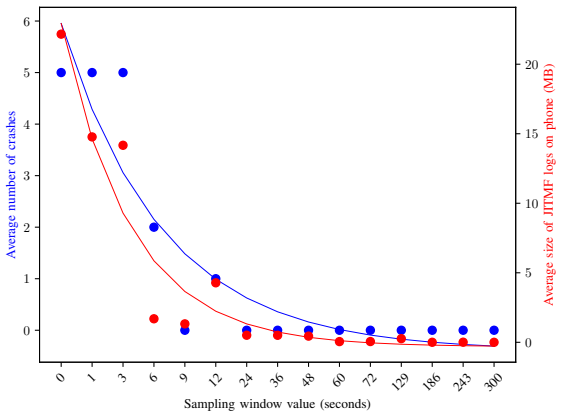
2) Results

Figures 7a,7c,7e present the results obtained when using *periodic sampling*, while figures 7b,7d,7f 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).

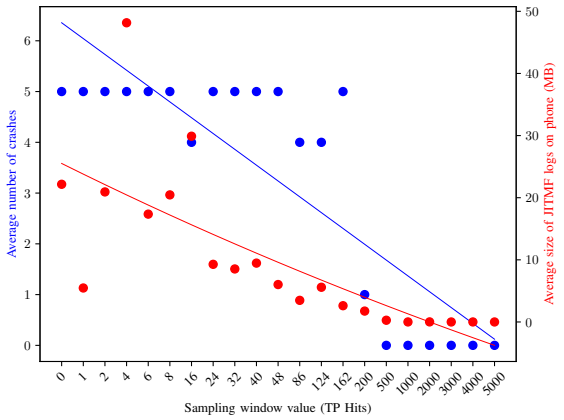
As expected, overall across both sampling methods, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

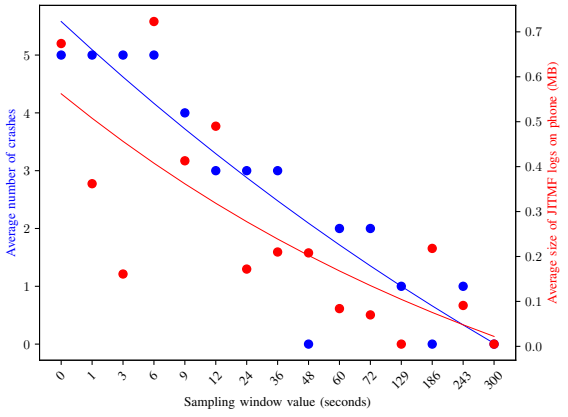




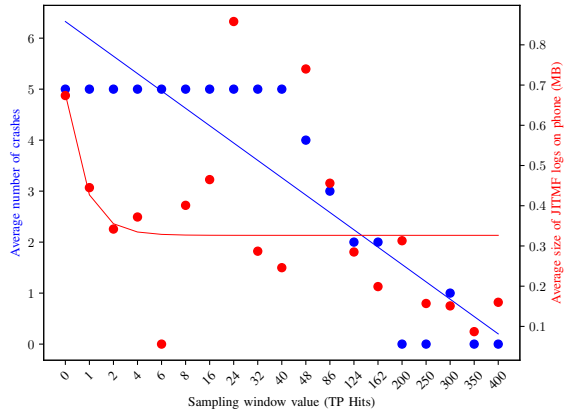
(a) Pushbullet: Average Number of App crashes and Storage requirements (MB) against increasing periodic sampling window (in seconds).



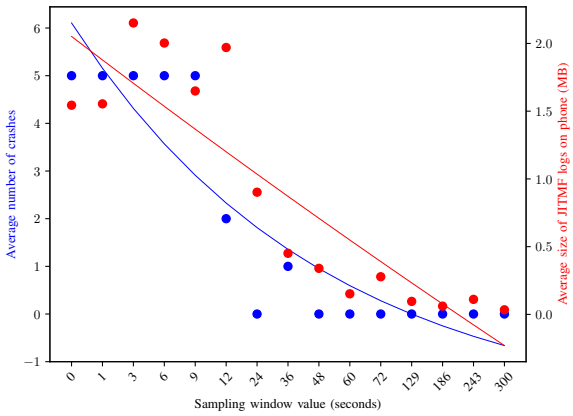
(b) Pushbullet: Average Number of App crashes and Storage requirements (MB) against increasing systematic sampling window (in number of TP hits).



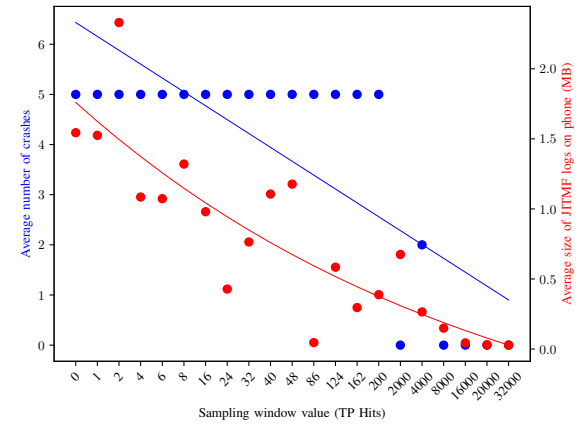
(c) Telegram: Average Number of App crashes and Storage requirements (MB) against increasing periodic sampling window (in seconds).



(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 7: Sampling results obtained for Signal, Telegram and Pushbullet

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

notice that with the increase in sampling window, the number of crashes and storage required, 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 sampling window, the output is still produced and, JIT-MF is still effective in dumping evidence objects in memory. Table 7 shows the percentage of events that were retrieved, for the sampling window where it is evident from the graphs presented in Figure 7, that a plateau has been reached. Results from this table show that, even in the cases where a large sampling window is adopted, the majority of ground truth events are still collected by MobFor, demonstrating that the Garbage Collector is working in MobFor's favor. This bodes well for the applicability of MobFor in a realistic scenario, especially when considering the improvement to the app's performance in terms of the number of crashes.

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 usage of storage resources.

Results from Figure 7 show that both sampling methods were equally successful in reducing significantly 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 in terms of the implementation effort required for each method, as well as the likelihood of each sampling method to fall prey to adversarial tactics.

Implementation effort. Periodic sampling requires a few more computational steps, as it is required to obtain current time, translate it to seconds and then check 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, well periodic sampling can be implemented even with no prior knowledge of the app's usage pattern or understanding as to how the app functions, the same cannot be said for systematic sampling. The latter method of sampling 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 the app itself functions. Therefore a preliminary exercise has to be done before applying this sampling method, whereby the average number of trigger point hits in a range of time 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 our implementation. In the case that the random value can be determined by an attacker, 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. However, this may be circumvented by an attacker by monitoring periods of inactivity and carrying out an attack at those specific instances, which in the case of stealthy attacks, is typically the case. Even in such scenarios, when trigger point callbacks are purposefully evaded to prevent memory dumps, eventually trigger points will be hit even if this occurs while the device owner is making use of the app. As previously discussed, when this occurs, the garbage collection functionality can still work in the investigator's favour and potentially dump missed objects.

The methods adopted in this experiment can give JIT-MF driver developers an indication of the storage requirements for a particular sampling window size, as well as the risks and implementation effort involved. This allows driver developers to make an informed decision as to which parameters suit their particular scenario.

3) Threats to validity

Enabling sampling allows MobFor to reduce the amount of storage occupied. However, if the device is already running low on storage, then a JIT-MF-based tool would run into the same issues of evidence availability as the baseline sources. Furthermore, the results obtained are concerning specific app usage patterns that we defined. Namely, in our experiments, the usage pattern involved the sending and loading of text messages, within the apps considered. Other possible usage scenarios include the transfer of complex media types as well as different usage patterns e.g. long time of inactivity or influx of incoming messages. While we envisage that the respective object of such media is located in memory as well and may be dumped just the same, further code comprehension may be required to fully understand how these objects can be carved and parsed.

The experiment was enabled through the use of Android emulators, to facilitate automation. However, the emulated devices used were unrooted, to ensure that no additional privileges given in the emulated setting contribute towards results that cannot be obtained in a realistic setting given stock Android devices.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

TABLE 7: % 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 app and incident scenarios considered in this experiment are only related to messaging hijacks. Other apps may still be subject to issues that require further experimentation.

C. MOBFOR IMPLEMENTATION

The results obtained in the previous subsections demonstrated that the first two objectives of MobFor can be met by: i) Using black-box trigger points to lessen reverse-engineering efforts required while still accurately dumping evidence objects from memory and ii) Using sampling methods with sampling windows suited to the scenario and device in question to lessen performance burdens while the app is running. This has been demonstrated in the context of JIT-MF drivers tailored for messaging hijack scenarios, targeting Pushbullet, Telegram and Signal.

MobFor is implemented using a subset of the Frida runtime, and JIT-MF drivers are implemented as Javascript code for Frida's Gadget shared library, allowing MobFor to function on a non-rooted device. The JIT-MF drivers targeting specific apps are modular and additional drivers can be added to MobFor as needed. The output produced by MobFor is in JSON format, a popular format among many forensic analysis helper tools, including Timesketch [42], and is stored to external storage `sdcard`. Further processing is required on the files outputted by MobFor to remove possible redundant duplicate data and to carry out parsing and carving of objects dumped if either of these is set to offline in the JIT-MF driver. The resulting output contains data as shown in Listing 9, where the metadata of a critical object retrieved from memory can be seen in the nested JSON *object* key. The *time*, *event* and *trigger_point* values refer to the time the trigger point that dumped the object, was hit, the type of event (depending on the JIT-MF driver used) and the trigger point type (*native* | *rt*), respectively. Figure 8 shows how Listing 9 is displayed when using the forensic analysis timeline tool Timesketch.

Telegram and Signal are popular messaging apps that are open source. Therefore, they did not present any challenges concerning instrumentation. Pushbullet is a closed source and a decompilation process revealed obfuscated code with multiple processes, each of which could be instrumented, unlike the other apps that only ran one process. It was noted that Pushbullet uses a subprocess that runs in the background handling incoming and outgoing messages. Therefore the $Process$ parameter in the JIT-MF driver was set to the Java class that initiates this subprocess, subsequently loading the Frida gadget library inside the process memory of the Pushbullet's subprocess, which handles the syncing of messages. These apps were able to function normally even when instrumented with JIT-MF drivers and presented no obstacles in the form of code integrity checks.

Finally, to maintain a chain of custody, a log is kept as shown in Listing 10 and updated with every command that is sent to the device mirroring the steps in Figure 1.

D. INSTALLING AND USING MOBFOR

MobFor is open-source is publicly available[15] and a full installation guide[16] as well as the supporting resources are provided. As explained in Figure 1, the JIT-MF workflow starts by first identifying target users and apps, then proceeds to instrument the identified apps. This is reflected in the implementation of MobFor, whereby as shown in Figure 9 and 10, investigators can choose to initialise a workflow that allows them to instrument an attached device with one of the available JIT-MF drivers. MobFor also caters for the collection of the evidence gathered on the device (Step 3 in Figure 1) and provides the output in a way that can be used for further timeline analysis, e.g. via Timesketch as shown in Figure 8.

By meeting all the objectives set out for a JIT-MF tool at the beginning of this section, we addressed the challenges presented by ***RQ2*** and build MobFor, a JIT-MF equipped with corresponding drivers that cater to messaging hijack attacks. We show this in an emulated setting involving Pushbullet, Telegram, and Signal and demonstrate that: i) Through the selection of black-box trigger points we can significantly reduce the required reverse engineering efforts, and ii) By using sampling, a JIT-MF tool can function successfully while being of minimal burden to the running app.

VII. USING MOBFOR IN A REAL-WORLD INVESTIGATIVE SCENARIO

With MobFor implemented, we now aim to demonstrate how MobFor can complement existing state-of-the-art digital forensics tools in a real-world incident response setting to address the question ***"In what ways can MobFor complement existing forensic tools, in a realistic setting?" (RQ3)***. To this end, we conduct an experiment involving a recently discovered stealthy Android worm called WhatsApp Pink

[15]https://gitlab.com/mobfor/mobfor-project
[16]https://mobfor.gitlab.io/mobfor-pages/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Listing 9: Sample of MobFor output from a Signal messaging hijack scenario after deduplication process.


```
1 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363829423", "message_id": "9", "text":
      "Noise_Qpj2hrJsI2bGaDrfMMujjesvC4a7nrS50IsuW", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX", "from_id": "RecipientId::1", "from_name"
      : "null", "from_phone": "+356XXXXXXXX"}}
2 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363816560", "message_id": "8", "text":
      "Noise_rTxVvTn2n5Is4JMtXPRcg76PyxEs0QtADXLf4WmjsQBU6EJf0YRpzom0V1409I8GpiNWeNGD7OR8Z9kUZL", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356
      XXXXXXXX", "from_id": "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
3 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363571590", "message_id": "7", "text":
      "Noise_BYSXwbV8wM70x9gTO0aCbd7FgiS6cSKJEOfCPQ8NTgJWsu4XYIAtLeKqbcJmqG5aWKA97KMI", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX",
      "from_id": "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
4 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363558600", "message_id": "6", "text":
      "Noise_8EqpPYRCcKcPP2NJjZWn2z8ENcOPokhfeHUjJOFrvRqGVqP5uGPOQND1", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX", "from_id":
      "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
5 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363544539", "message_id": "5", "text":
      "Noise_sgdyg4Jxtyvw05AumxJpN9GfSQOT9pRCDeZ8SptXWihp5hr3RfL86H77wfgO2hlc7Oz5HnVPIo0eN", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX",
      "from_id": "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
6 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363531629", "message_id": "4", "text":
      "Noise_P0n9mzE7HT19lgTT0EZQlSj5aJBurU7SOtlmggkbDXhuBZFu4Kn9HW40yAoCwkNX", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX", "from_id":
      "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
7 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363518692", "message_id": "3", "text":
      "Noise_XNPvOs7gDxEflYtbr3kgE9kALDyoSSfigD6WAOUPlt2DK5d2KvG1tFdnVZ", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "+356XXXXXXXX", "from_id":
      "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
8 {"time": "1616363830", "event": "Signal Message Sent", "trigger_point": "native", "object": {"date": "1616363504554", "message_id": "2", "text":
      "Noise_vaQNUiyKcJIUjXqJPnjbRpFKsrBdgYLt3YZkYdlYlPhvONhFucXwBP4uuaNCy5M2sqN0as9nqCDeFmc9FfUGKN98h6CMHXOpsA1", "to_id": "RecipientId::2", "to_name": "null",
      "to_phone": "+356XXXXXXXX", "from_id": "RecipientId::1", "from_name": "null", "from_phone": "+356XXXXXXXX"}}
```

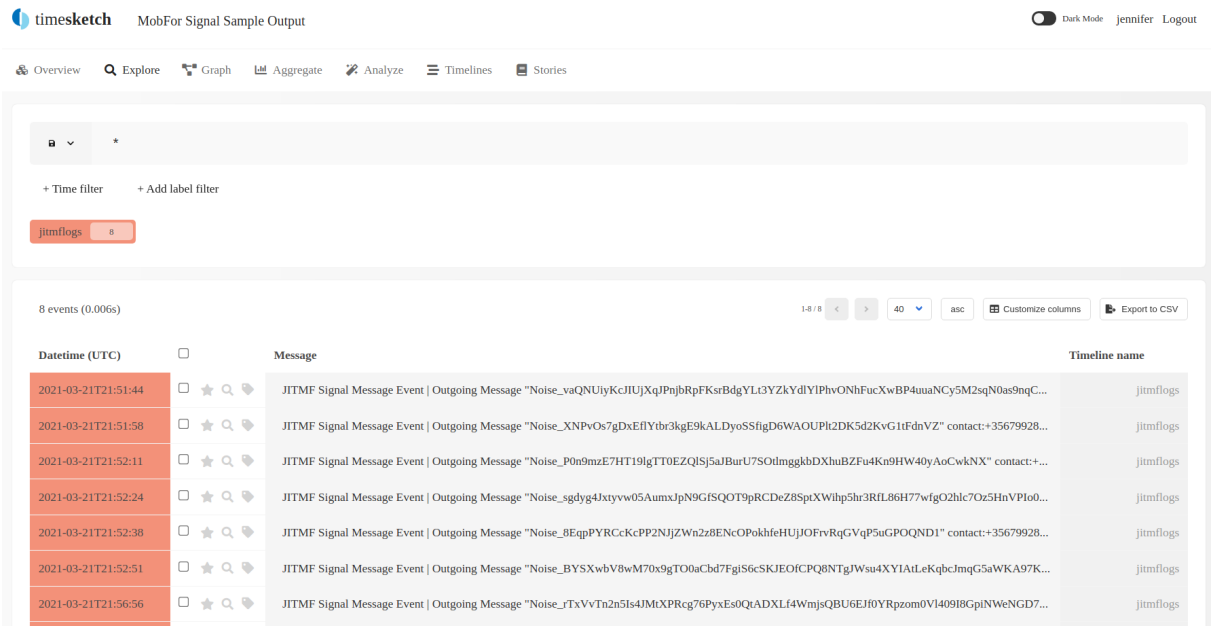



FIGURE 8: Sample of MobFor output from a Signal messaging hijack scenario as displayed in Timesketch.

[69] (a messaging attack involving WhatsApp) and make use of two digital forensics tools: Belkasoft Evidence Centre X[17] and MSAB XRY[18], along with MobFor. Since the context involves an app that was not considered in our previous experiments, we also create a new JIT-MF driver for MobFor, targeting messaging hijack attacks involving WhatsApp, thereby allowing the opportunity to study the entire JIT-MF process.

A. WHATSAPP JIT-MF DRIVER DEFINITION

The implementation of the JIT-MF driver for WhatsApp followed a similar approach to that used to generate drivers for Pushbullet, Telegram and Signal, with minor modifications that are specific to WhatsApp as shown in Listing 11.

[17]https://belkasoft.com/x
[18]https://www.msab.com/products/platforms/#office

Listing 11: JIT-MF driver for WhatsApp.

```
1  Driver_ID: WHATSAPP_MSG_HIJACK
2  Scope: <whatsapp, msg-hijack>
3
4  /* Attributes */
5  Collection_method: online
6  Parsing_method: offline
7  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()>>}
8  Sampling_method: sampling_predicate()
9  Log_location: "/sdcard/jitmflogs"
10 Globals:{<timestamp,>}
11 Evidence_objects: {} // the evidence object is added
       at runtime, since it is an argument to the
       trigger point
12
13 /* Exposed interface */
14 bool init (config) {
15   for entry in Triggers:
16     if entry[1] == native:
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

Listing 10: Snippet of Chain of Custody Log.


```
[{"transactionId": 8, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Device Attached", "lowLevelEvent": "Device emulator-5554 attached", "evidence": "", "evidenceHash": "", "timeOfTransaction": 1632470927},
{"transactionId": 9, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-List user-installed apps versions", "lowLevelEvent": "adb shell dumpsys package org.telegram.messenger | grep versionName | awk -F '=' '{ print }'", "evidence": "5.12.0", "evidenceHash": " ", "timeOfTransaction": 1632471053},
{"transactionId": 10, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-List user-installed apps", "lowLevelEvent": "adb -s emulator-5554 shell pm list packages -3\"|cut -f 2 -d \":", "evidence": "org.telegram.messenger", "evidenceHash": " ", "timeOfTransaction": 1632471053},
{"transactionId": 11, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-Get path for selected apk", "lowLevelEvent": "adb -s emulator-5554 shell pm path org.telegram.messenger | awk -F':' '{print }' | tr -d 'r'", "evidence": "PATH of apk on device: /data/app/org.telegram.messenger-SNgL41Mr_-kwpvBdA-wh9w==/base.apk", "evidenceHash": "", "timeOfTransaction": 1632471072},
{"transactionId": 12, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-Pull chosen app from device", "lowLevelEvent": "adb -s emulator-5554 pull /data/app/org.telegram.messenger-SNgL41Mr_-kwpvBdA-wh9w==/base.apk /root/.mobfor/workflow_1/original_apk/", "evidence": "Original apk location: /root/.mobfor/workflow_1/original_apk/org.telegram.messenger.apk", "evidenceHash": "fac646e04d73f20790b68eb75109c5fc9664e9134a915a3cfc3a7d1a51280b1e", "timeOfTransaction": 1632471073},
{"transactionId": 13, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-Patch and Install app", "lowLevelEvent": "jitmf patchapk -apk_src=/root/.mobfor/workflow_1/original_apk/org.telegram.messenger.apk -hook=/jit-mf/resources/drivers/telegram_driver.js -dev_id=emulator-5554 -jitmf_dir=/root/.mobfor/workflow_1/generated_apks --install", "evidence": "Path to logfile: /root/.mobfor/workflow_1/logs/runtime_log_20210924_111300.log", "evidenceHash": "69b06b9740fca238249e5c1d1cb87eb159922221304b42e3a18bd14b8e8b826c", "timeOfTransaction": 1632471224},
{"transactionId": 14, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Forensic Enhancement-Installed app", "lowLevelEvent": "jitmf patchapk -apk_src=/root/.mobfor/workflow_1/original_apk/org.telegram.messenger.apk -hook=/jit-mf/resources/drivers/telegram_driver.js -dev_id=emulator-5554 -jitmf_dir=/root/.mobfor/workflow_1/generated_apks --install", "evidence": "Path to forensically enhanced apk: /root/.mobfor/workflow_1/generated_apks/org.telegram.messenger.jitmf.apk", "evidenceHash": "6c4b3019e4ecf203b3c60c9a88c68aac8e467a62cb8f18958a933187d434bceb", "timeOfTransaction": 1632471225},
{"transactionId": 15, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Device Attached", "lowLevelEvent": "Device emulator-5554 attached", "evidence": "", "evidenceHash": "", "timeOfTransaction": 1632471511},
{"transactionId": 16, "workflowId": 2, "workflowDescription": "workflow_1", "investigator": "jennifer", "deviceId": "358240051111110", "highLevelEvent": "Data Collection-Pull evidence from device", "lowLevelEvent": "adb -s emulator-5554 shell find /sdcard/jitmflogs -iname '*.mobforlog' | tr -d '\\015' | while read line; do adb pull '$line\\' /root/.mobfor/workflow_1_locard/jitmf_output/raw; done;", "evidence": "Path: /root/.mobfor/workflow_1_locard/jitmf_output/raw/merged-logs-uniq.jitmflog", "evidenceHash": "Hash: bf89e609fc43034be7a378ee99cc9ca6e2c9a91a9bdf28aa8b486fbbd6e220f2", "timeOfTransaction": 1632471620}]
```

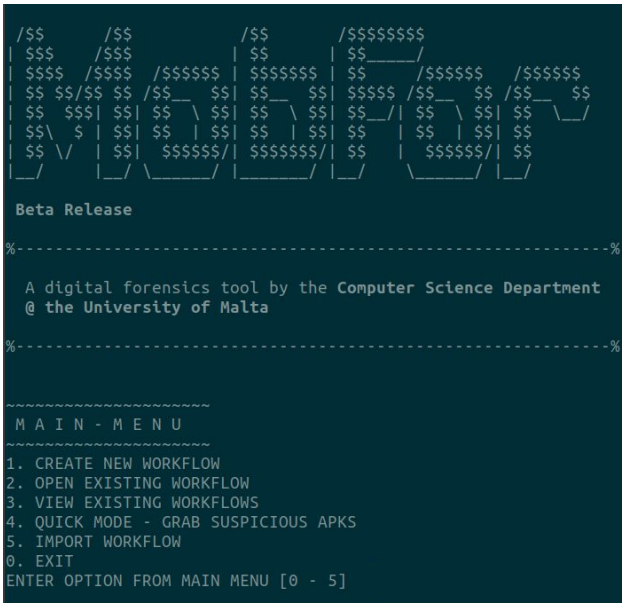



FIGURE 9: MobFor Main Menu.

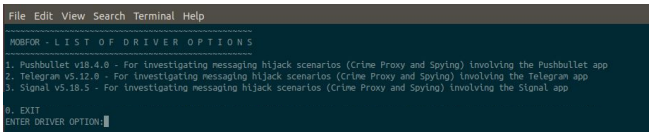


FIGURE 10: MobFor JIT-MF driver selection screen.

```
      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);
}
```

```
65
66 @OFFLINE
67 [<field,value>,...] parse_content_values_offline(at) {
68   if Collection_method == online:
69     current_time = get_time();
70     ContentValuesObject = object starting from at;
71     ContentValuesObject_strings = base64 decode
           strings in ContentValuesObject;
72
73     message_content = last string in
           ContentValuesObject_strings;
74     message_id = "";
75     message_date = Globals["timestamp"];
76
77     if 'SendE2EMessageJob' in
           ContentValuesObject_strings:
78       recipient_phone_number = match for regex search
             "@s.whatsapp.netsr[0-9]+\\n" in
             ContentValuesObject_strings;
79       recipient_id = "";
80       recipient_name = "";
81
82       sender_phone_number = owner phone number;
83       sender_id = "";
84       sender_name = owner name;
85       event_additional_info = "sent";
86     else if 'SendReadReceiptJob' in
           ContentValuesObject_strings:
87       sender_phone_number = match for regex search "@s
             .whatsapp.netsr[0-9]+\\n" in
             ContentValuesObject_strings;
88       sender_id = "";
89       sender_name = "";
90
91       recipient_phone_number = owner phone number;
92       recipient_id = "";
93       recipient_name = owner name;
94       event_additional_info = "read";
95     }
96
97       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}}");
98
99       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>>];
100  else:
101    // use an hprof parser to parse object_j at the
           given offset from memory dump;
102  }
103 }
104
105 [<field,value>,...] parse_content_values_for_timestamp(
       at) {
106   ContentValuesObject = object starting from at;
107   Globals["timestamp"] = ContentValuesObject["
        sort_timestamp"];
108   return [<>];
109 }
110
111 bool sampling_predicate(thread_context) {
112   return true;
113 }
114
115 /* Helper function */
116 datetime get_time(){
117   return current time;
118 }
```

Trigger Point Selection. As with previously created JIT-MF drivers, we aim to choose a black-box trigger point, based on the known basic functionality of WhatsApp. In the case of WhatsApp, we know that messages are *stored* and *loaded* from a local SQLite database. Even though WhatsApp is closed-source, it still makes use of Android's *SQLite-Database* API (see Table 1), providing a convenient avenue for trigger points while avoiding further app reversing. The `android.database.sqlite.SQLiteDatabase .insert()` method was chosen as trigger point for this purpose. This trigger point falls under the black-box set of trigger points, as part of the *Android & 3rd Party APIs* category.

Evidence Object Selection. The evidence object indicating a possible attack step is a WhatsApp message. Given the publicly available documentation related to the chosen trigger point[19], we are able to conclude that the final parameter of this function (*android.content.ContentValues*) contains a reference to the object of interest. Therefore, no additional code comprehension/reverse engineering effort was required in determining the message object of interest from obfuscated WhatsApp bytecode. In this case, the actual evidence object is added to the list of $Evidence\_objects$ when the trigger is hit and the callback is executed (see lines 30-31 in Listing 11) and is passed onto the $dump\_rt\_object$ method to be carved and parsed. Upon further inspection of the object, we notice that the object is Base64 encoded and once decoded, the message content, recipient and sender phone numbers can be parsed. Since this requires a fair amount of processing, parsing of the relevant metadata is done offline (see lines 67-103 in Listing 11). A crucial parameter is missing: the timestamp associated with a message. We notice from preliminary runs that prior to an `insert()` an SQLite table `update()` is executed, updating the timestamps of messages sent/received. Therefore, a secondary trigger point (line 7) is added to determine the timestamp of a message being `insert()`-ed. Finally, the contents of the object retrieved also provide information on the type of event that the object is related to (lines 85, 94), which we append to the event type previously defined, related to the scope of the driver.

Sampling Method. Preliminary observations were made to determine the sampling window range for WhatsApp. Specifically, we carried out an exercise to determine the TP hit count over a period of time, where activity was generated by simulating the sending and receiving of messages. As explained in the previous experiment, this is a precursor to setting a value for the systematic sampling method.

The selected trigger point was only invoked once per message sent; that is, whereas with other tested apps, the sending of 5 messages invoked, at worst, 32,000 TP hits,

[19]https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

with WhatsApp only 5 TP hits were incurred. This is likely because the trigger point selected in WhatsApp's case falls under the *Android & 3rd Party APIs* category rather than the *Native RT* category, which, as explained in Section VI-A2, may be less resource-intensive. To this end, no sampling was deemed necessary.

Code integrity checks. While no issues were encountered when instrumenting Pushbullet, Telegram, and Signal with JIT-MF drivers, WhatsApp presented an additional challenge. Along with having obfuscated code, WhatsApp also adopts anti-repackaging techniques in the form of app signature checks using `getpackageinfo()` as well as MD5 checks of the `classes.dex` file, which aim to prevent users from using a modified version of WhatsApp. In this case, these were standard checks that were easily bypassed by: i) hooking into the relevant functions performing the app signature checks and returning the appropriate value and, ii) using an alternative static library injection method that avoids modifying `classes.dex`. However, more advanced checks adopted by different apps could present a tougher challenge.

B. INCIDENT SCENARIO

The assumed scenario involves a target victim who is a high-profile employee at their workplace and has had their WhatsApp app on their Android device 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 messages with a download link to the malware itself akin to a message proxying attack.

WhatsApp Pink. Once this malware is installed, it requests a number of permissions. Namely, it requests for the *Notification Access* permission, which in conjunction with Android's Direct Reply action available in Android, 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 optimization, which allows it to run in the background and prevents the system from killing it off for any reason.

Figure 11 illustrates how WhatsApp Pink operates. The malware runs in the background and waits for a 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 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 [69].

Further malware hardening for stealth. While WhatsApp Pink already does its best to conceal its actions, even going a step further to hide the app's icon from the home screen, the propagated message by the malware is still visible in the chat. To further increase the level of stealth, we enhance the malware so that the message sent by WhatsApp Pink is deleted from the owner's phone, leaving no visible trace of the malware's actions, as is shown in Figure 12. This can be done by misusing the `adb` attack vector via the `AndroidViewClient`[20]. As a final step, we misuse the `adb` attack vector to uninstall WhatsApp Pink, and attain maximum stealth by leaving no trace of the malicious app and hence reducing further its forensic footprint.

C. 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 the collection and parsing features, enabling an incident responder to analyse the forensic timeline produced through the available tools. To demonstrate how MobFor can contribute to existing digital forensics tools in a realistic setting, such as the incident scenario described, we make use of two such tools: Belkasoft Evidence Centre X and MSAB's XRY 8. For each of the tools (including MobFor), we define the configuration used for incident response in Table 9. The table describes the sources gathered during the collection phase, the parsing tool used, and the tool used to generate a forensic timeline for analysis.

Limitations of the forensically enhanced WhatsApp. Although not publicly disclosed, one way how Belkasoft and XRY (as well as other tools) can collect private data, typically containing decrypted database files on disk, is through the use of `adb backup` after an app downgrade. Although recent versions of WhatsApp have their custom `BackupAgent`, older versions did not. Through the inspection of open source tools like WhatsApp DB Extractor[21] [70], [71], we noticed that one can uninstall a version of WhatsApp on their phone without deleting private app data containing texts using: `adb uninstall -k com.whatsapp`. A specific older version of the app can then be installed, whose Manifest file does not include a mention of the custom `BackupAgent` and much less

[20] https://github.com/dtmilano/AndroidViewClient
[21] https://github.com/EliteAndroidApps/WhatsApp-Key-DB-Extractor

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

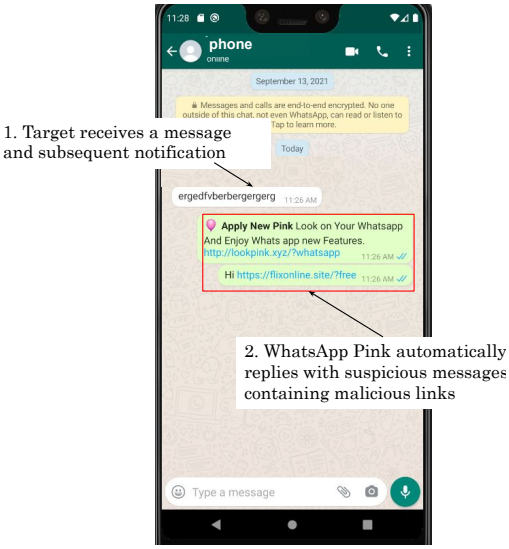


FIGURE 11: WhatsApp Pink Attack steps.

TABLE 8: Forensic Tools used.

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

TABLE 9: Forensic Analysis Configurations.

Incident Response Step	MSAB XRY Configuration	Belkasoft Configuration	MobFor Configuration
Collection	• XRY Agent • Additional forensic sources	• Belkasoft Agent • Additional forensic sources	• MobFor Forensic Sources
Parsing	XRY XAMN	Belkasoft	MobFor output parser
Analysis	Timesketch		

have an implementation for it. Therefore, after the app downgrade, an `adb backup` can be executed, collecting all the data that was previously unattainable due to an update in the app's `BackupAgent`. 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 app carries a different signature than the original one, as shown in Figure 13. The partial uninstallation of the enhanced app and the mismatch of the older app's signature cause Android to produce an error, preventing the older app from being installed. However, since MobFor already statically instruments the app to include the invocation of Frida Gadget, we further modify the Manifest with `debuggable=true`, allowing private WhatsApp files to be collected. The result is equivalent to the `adb backup` method.

Forensic Tool Setup. While the version of tools used were trials, this did not, to our knowledge, impact the results of our experimentation in any way. Most of the functionality withheld from trial versions is related to the availability of rooting exploits and iOS features. Since our scenario involves Android OS and the device belongs to a victim who intends to continue using it, rooting is a non-viable option. Therefore these features were not required anyhow. There are also some limitations with regards to the amount of data that could be exported from the findings when using Belkasoft. That said, all of the findings are still made available, at least through the user interface.

Both XRY and Belkasoft were configured in a similar fashion. An agent-based collection was used by each respective tool, targeting logical collection (see Figures 14 and 15. Whenever a collection step required rooting, it was skipped. Furthermore, each tool also accepts additional data sources that the analyst has in their possession to support the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

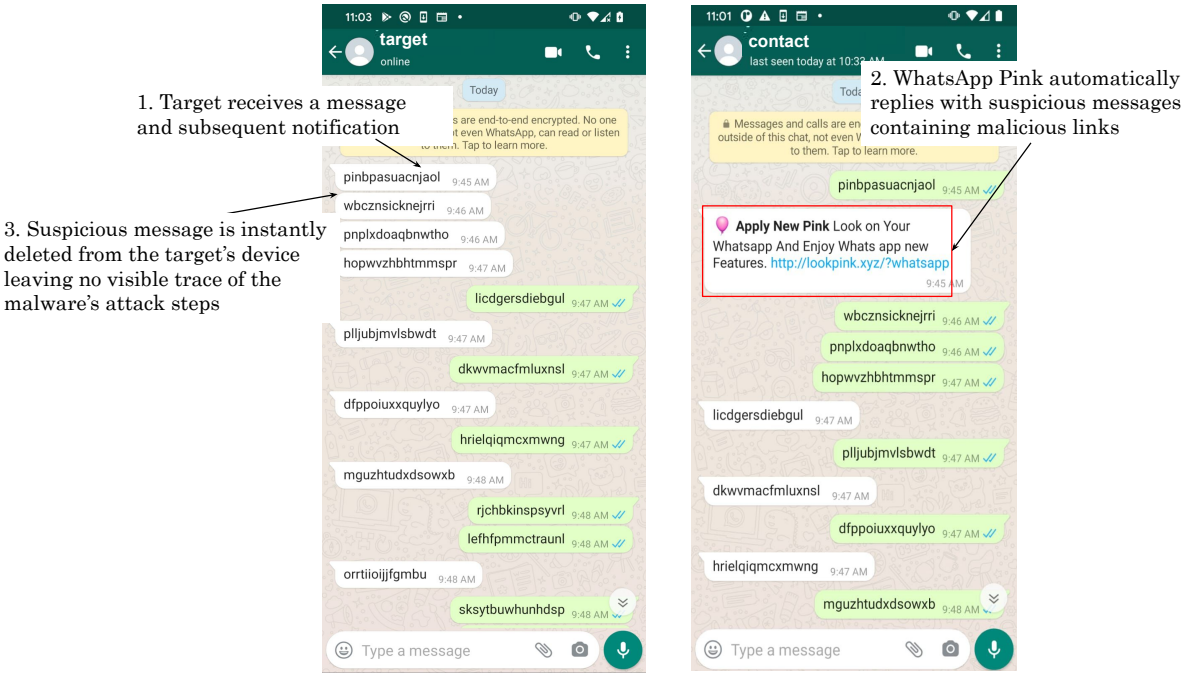


FIGURE 12: Enhanced WhatsApp Pink attack steps executed on Pixel 4 devices. The device on the left is the target device D running the malware and the device on the right is the contact device C from which a message was sent to the target.

findings gathered by the tool. Once the sources are gathered, each tool parses them and presents them in a way that they can be of use to the investigator, as shown in Figure 16. Figure 17 shows the parsed forensic artefacts gathered to be used for 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 `/data/data` folder, obtained through it's `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`.

D. MOBFOR EVALUATION

The incident scenario outlined in Section VII-B is assumed. MobFor is evaluated across three commercially available stock Android physical devices listed in Table 10, reflecting a range of device capabilities and limitations. Each of the devices has a forensically enhanced version of WhatsApp version 2.21.14.25 installed through MobFor and the additional JIT-MF driver defined in Listing 11. Finally, XRY and Belkasoft are used as explained in Section VII-C and Table 9.

We conduct this experiment three times, once for each device available. Each time, the enhanced WhatsApp Pink malware is installed and set up on the target device (D). Another phone (C) is set up emulating a contact that device D has. Figure 12 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 WhatsApp Pink malware installed on target device D immediately propagates the malicious link and subsequently 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 VII-C. The available forensic tools (including MobFor) are used as described in Table 9. Finally, the parsed forensic artefacts are then used to populate a forensic timeline using Timesketch.

E. RESULTS

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

Table 11 describes the findings of each forensic analysis configuration and device, with regards to the known attack steps executed (the ground truth). Overall, while none of the forensic tools' outputs individually 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 Figures 18 and 19. Belkasoft was able to retrieve the event of a message being sent. However, the message content was missing. While this event in itself is suspicious, without the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

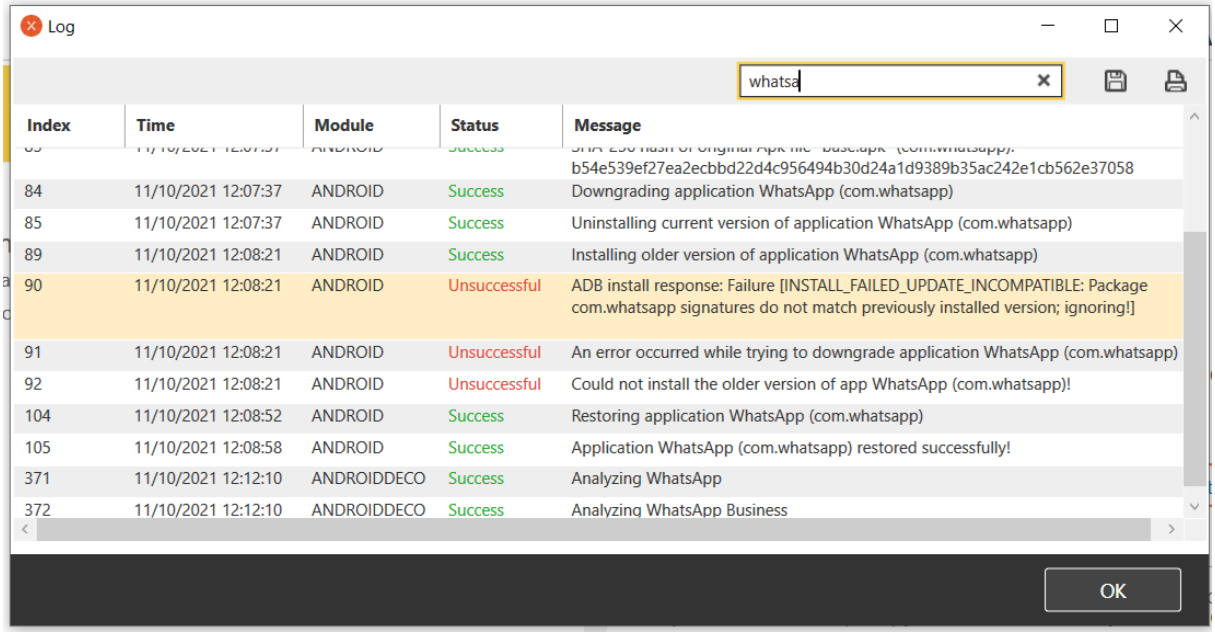


FIGURE 13: XRY Log output showing WhatsApp downgrade failure.

TABLE 10: 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

TABLE 11: Recovered events.

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

message content, it is not clear 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. We assume that since the content was missing in the WhatsApp databases, due to deletion, the event is not even displayed to the investigator to start with.

Additional Challenges. The Frida Gadget library is built on top of Android API; therefore, any changes to the API may require updates to the Frida Gadget itself. Such is the case with Android 11. To this end, the instrumented WhatsApp installed on the Samsung S21 device was instrumented with the latest Frida Gadget library.

Due to the awareness and publicity that WhatsApp Pink garnered in the last few months, the app has been flagged as malicious. The Pixel 4 device, in particular, required an extra step to accept the installation of the malicious app. That said, techniques like accessibility or overlay could have been used to conceal this step from the user further and maintain stealth.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

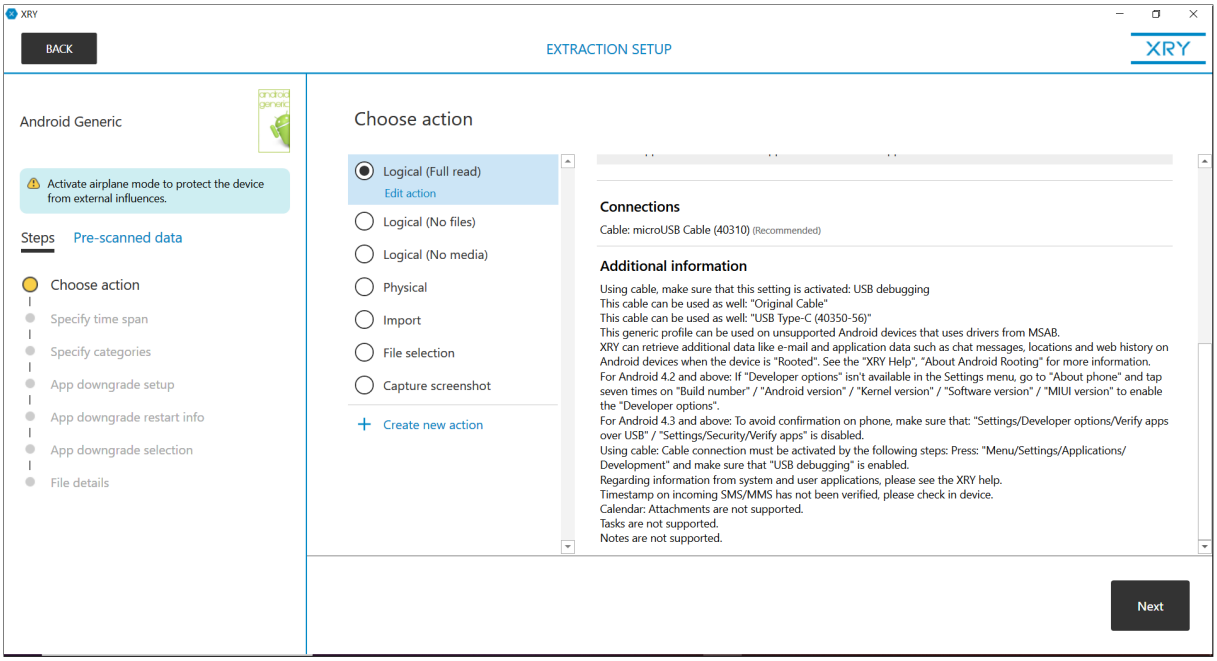


FIGURE 14: XRY Collection Method (1).

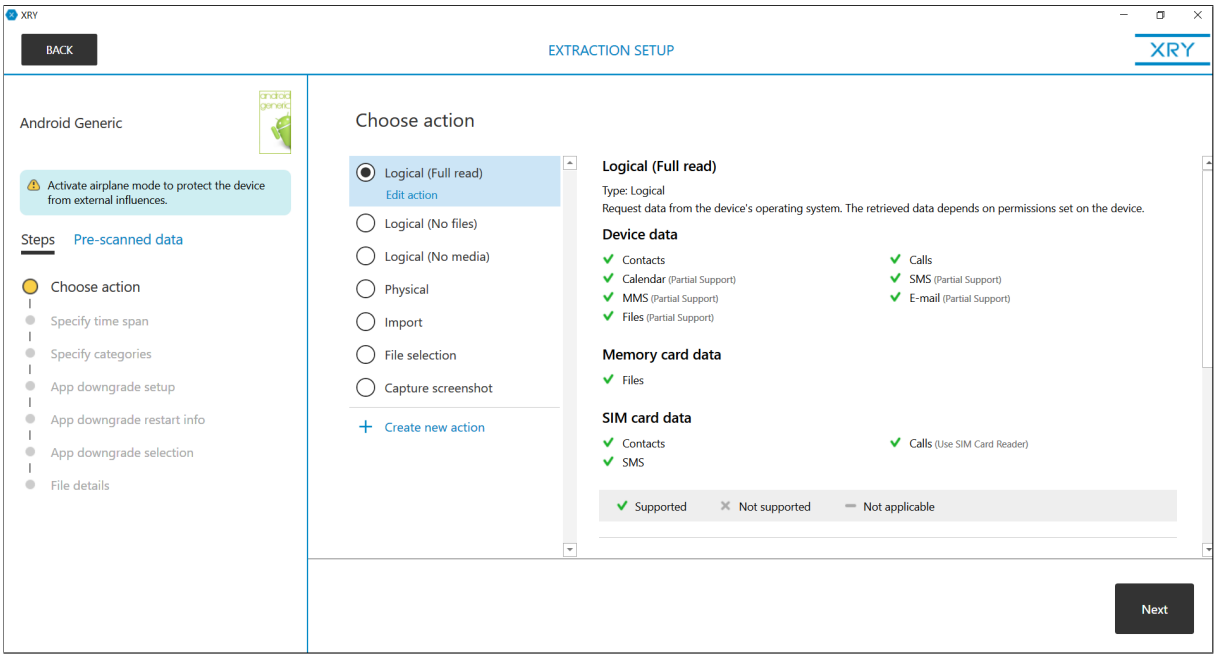


FIGURE 15: XRY Collection Method (2).

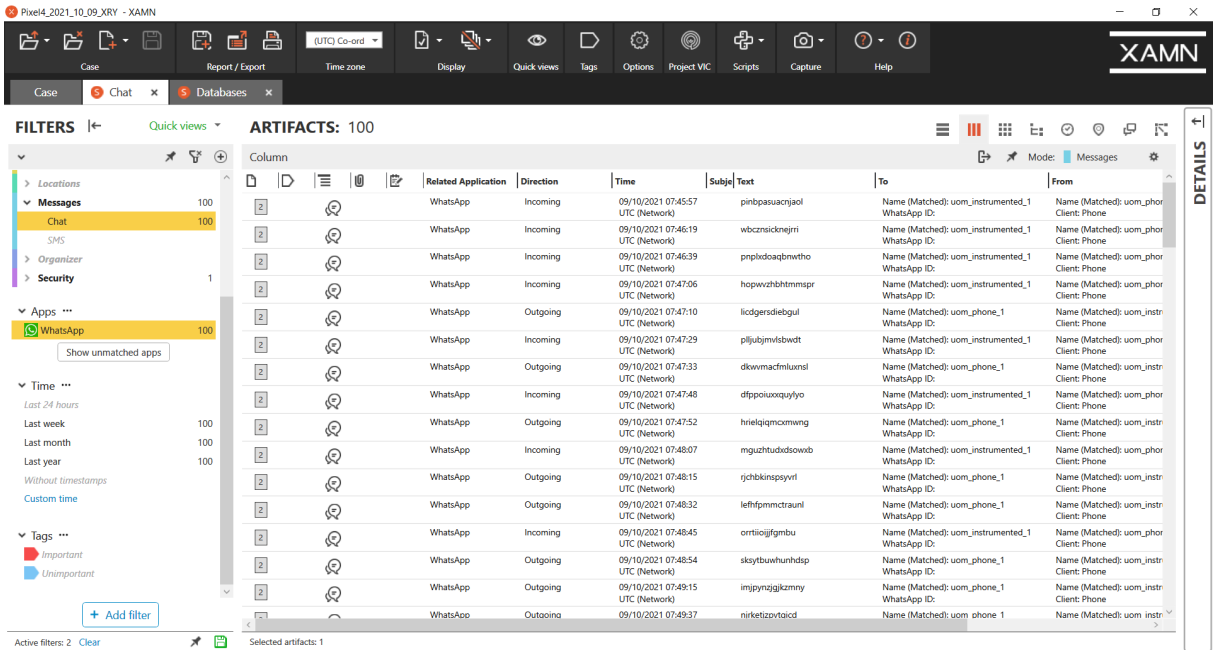


FIGURE 16: XRY Output containing sample WhatsApp messaging events.

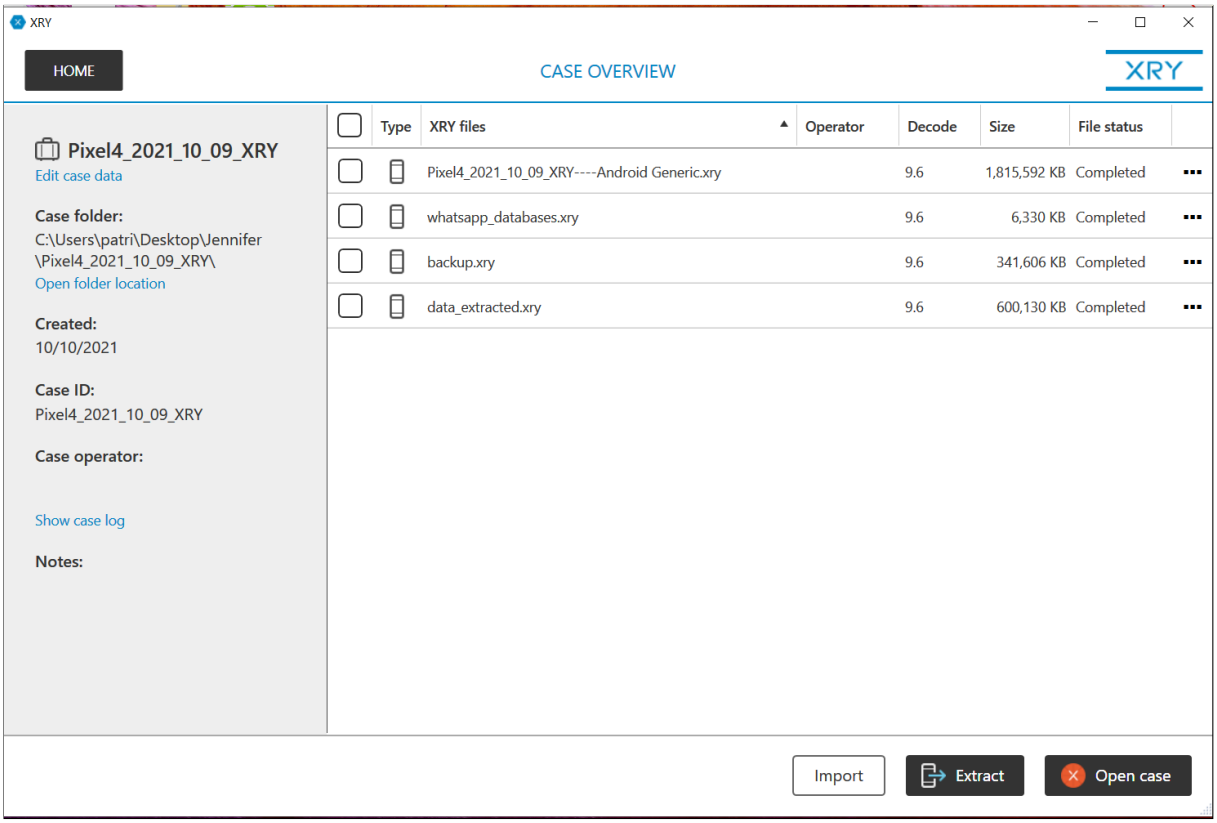


FIGURE 17: XRY Additional Forensic Sources.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

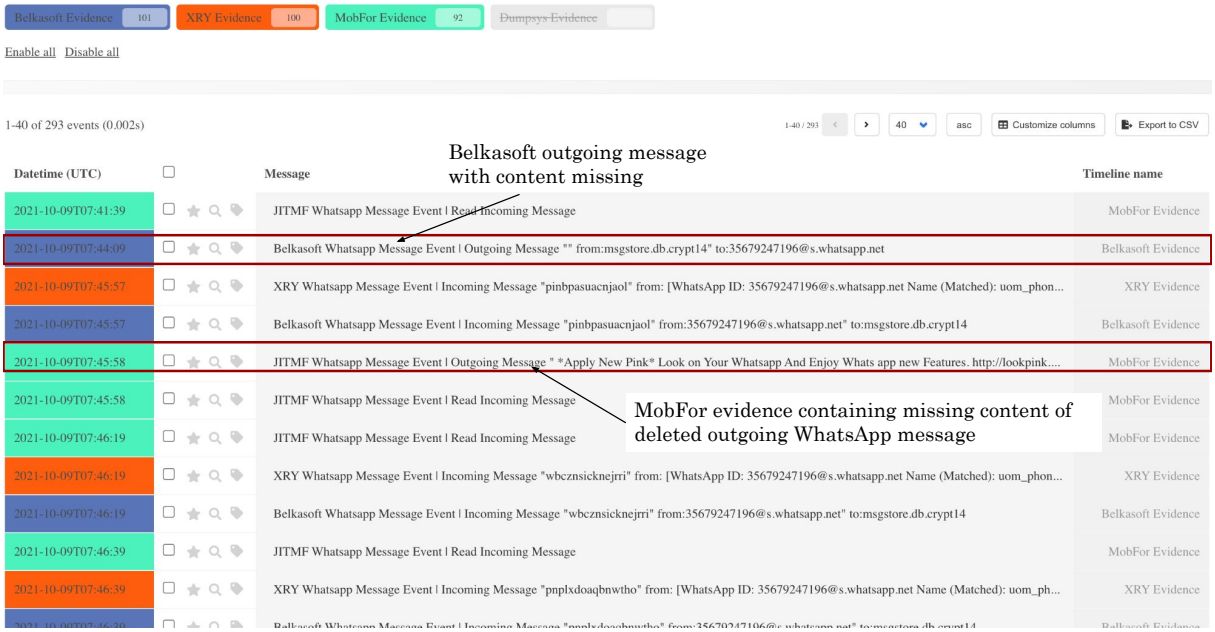


FIGURE 18: Identification of a suspicious event due to differences between forensic sources outputs related to the same event.

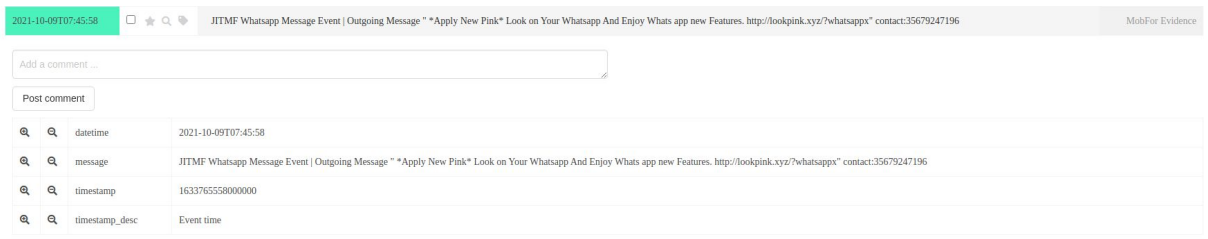


FIGURE 19: Additional metadata produced by MobFor in the case of a suspicious messaging event.

Forensic Analysis. The results in Table 11 are given with respect to the known message object that we are after, given that we are aware of the attack steps involved in WhatsApp Pink. However, in typical investigation scenarios, investigators do not have the luxury of knowing which specific forensic artefact might be helpful in an investigation. Our aim is that using MobFor; investigators can obtain all the possible artefacts needed to construct a more complete timeline. While this does not ensure that a case/incident is solved, as that is mainly dependent on the incident responder's manual analysis skills, this cannot be done without at first having all the evidence related to an attack at hand. Furthermore, while MobFor (through the use of the JIT-MF technique) can retrieve critical metadata in attack steps, other sources of evidence are still required to understand better the context involving an attack step.

We assume the role of an incident responder and use Timesketch 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 9, `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 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 (thirty minutes), of which only three events were related to the malware's activities. To this end, we initially start our investigation by focusing solely on the events produced by forensic tools in Table 9.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

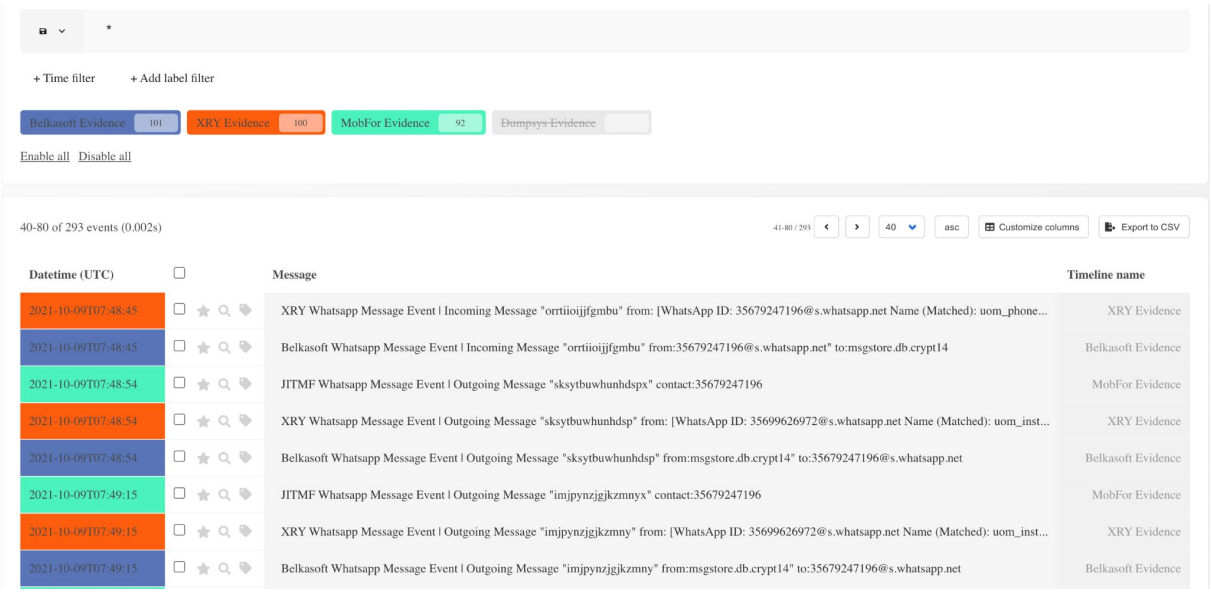


FIGURE 20: Combined timeline of events produced by Belkasoft, XRY and MobFor, in Timesketch.

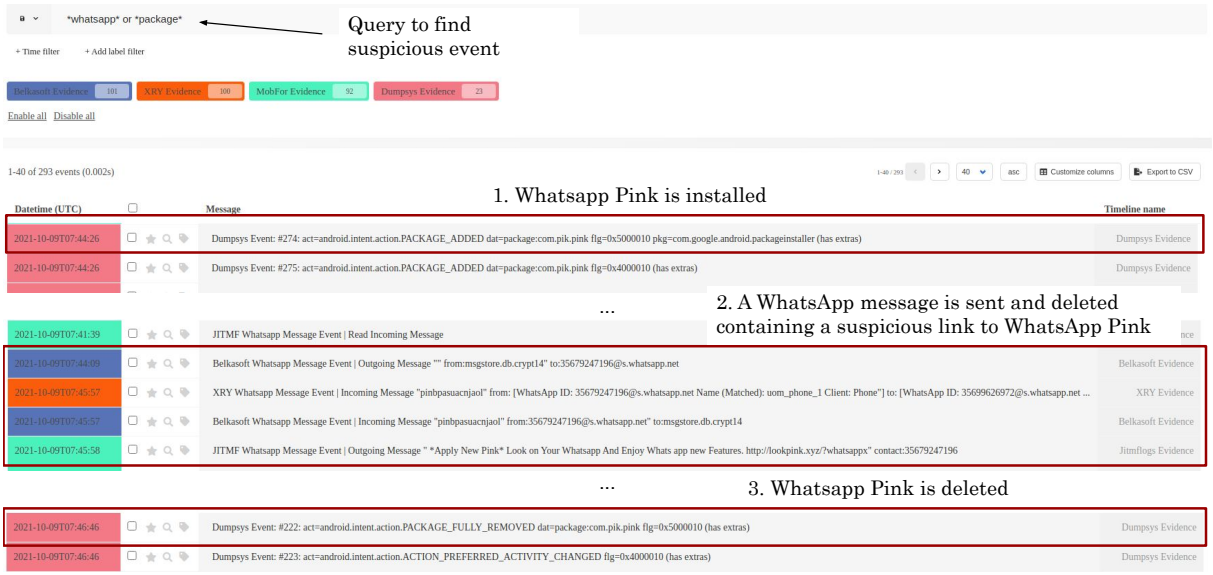


FIGURE 21: Recovered sequence of attack steps from Timesketch timeline.

Figure 20 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. In the timelines produced for the scenarios in this experiment, we notice this pattern with all the messages that are sent between devices D and C, except for one event. When focusing on this event, we notice that it is missing in the evidence generated by XRY. Even more suspiciously, its contents are empty in the evidence generated by Belkasoft. In contrast, MobFor output shows that the message content contains a link as shown in Figure 18. The discrepancy between the tools outputs' regarding the same event already suggests that this event is suspicious. We confirm that the message is not on the target's device, as can be seen in Figure 12, which allows us as investigators to conclude that the message has been deleted from the target's device.

To get a better understanding of whether or not the target or a stealthy malware carried out these actions, we widen our investigation to include evidence from `dumpsys`. Due to the large number of events produced by this source, we narrow our scope by creating a query to look for artefacts containing any data containing `whatsapp` data or `package`-related events. The set of steps executed by the malware is retrieved as shown in Figure 21. The steps show the malware being installed, the presence of a message that has since been

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

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, we conclude that 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 steer the investigation in the right direction.

VIII. DISCUSSION & FUTURE WORK

This research focused on exploring JIT-MF as a memory forensics technique and creating MobFor as a showcase JIT-MF tool that can be used in a realistic setting and other tools to aid in digital forensic investigations involving Android devices. Results from the experiments carried out demonstrate the following.

Stealthy attacks targeting Android leave a severely reduced forensic footprint when utilising state-of-the-art mobile forensics tools. Results from both XRY and Belkasoft (two state-of-the-art forensic tools) show that, even after collecting all the possible app data from the target device, stealthily deleted messaging events could either not be retrieved at all or were missing critical metadata (e.g. message content). The main limitation of such tools stems from their assumed context, in which the device owner is a potential perpetrator rather than the targeted victim of cybercrime and does his best to conceal any compromising evidence from investigators. Furthermore, such tools rely on the data generated and stored by the app itself. The scenarios presented in this study show that in the case of stealthy messaging attacks, none of the apps considered generate and store data related to the attack steps carried out through the misuse of benign apps. Therefore, in the case of stealthy attacks, the forensic sources collected by state-of-the-art tools are missing critical events related to the attack steps carried out.

Android memory forensics as an additional forensic source. The memory forensics approach taken by JIT-MF tools has been shown to complement existing state-of-the-art tools by providing additional forensic sources that contain missing evidence collected from memory. Results from the experiments conducted in this research demonstrate that JIT-MF is not only able to capture stealthy events by collecting evidence from memory, but the events captured also contain critical metadata that is missing from other state-of-the-art forensic sources that can fill in gaps in existing forensic timelines and hence help generate a more complete forensic timeline for investigators to work with.

A forensic readiness stage is needed. The scenario that we target is one whereby the Android device owner can be a potential target of a cybercrime rather than a perpetrator. In such cases, JIT-MF tools can be used to forensically enhance Android devices by instrumenting benign apps that could be leveraged in the case of an attack, with JIT-MF drivers that collect objects from their process memory while the app is running. This requires collaboration with the device owner, as the apps to be instrumented would first need to be uninstalled, then their instrumented counterparts would be reinstalled and logged into. This completes the device forensic readiness stage that JIT-MF requires and enables JIT-MF to function on stock Android devices without rooting.

While the collaboration of the device owner is a prerequisite for JIT-MF to work, their trust is not. Privacy concerns related to the public disclosure of the device owner's private data found in the dumped parsed evidence from memory, e.g. message contents, remain an important issue that needs to be addressed. Non-technical solutions that could alleviate such privacy concerns include the investigative party (e.g. investigators) signing non-disclosure agreements (NDA) that would prohibit them from publicly disclosing the contents of the evidence. There are also possible technical solutions from JIT-MF's end that can be considered in the future to ascertain the device owner further that the privacy of the evidence collected from their phones is respected. Searchable Symmetric Encryption (SSE) [72] can be adopted to allow the privacy-preserving forensic analysis of evidence by encrypting the dumped forensic evidence collected 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 or call data records (CDR) that identify texts to be suspicious, for instance, text format of URL addresses that are known to be propagated by specific malware or suspicious phone numbers found in CDR. Investigators can then flag suspicious events that may have been caused by a malicious actor without having had access to the private data belonging o the device owner.

Minimal code comprehension effort is required. Results show that while the timely dumping of evidence from memory is target app and investigative scenario-specific, this does not require extensive reverse engineering or code comprehension. The results show that effective trigger points can be established using black-box analysis, with the most effort required to identify critical data object types for decoding raw evidence objects.

Existing limitations with implementing JIT-MF in practice. While several implementation challenges were addressed in this study, allowing for the creation of MobFor — a tool shown to be functional and provided additional value in a real-world setting — several challenges remain.

App developers use anti-repackaging measures involving code integrity checks to hinder the customisation of apps. Given that JIT-MF operates by customising apps in a post-deployment manner, anti-repackaging measures affect the amount of reverse engineering required to develop JIT-MF

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

drivers that can bypass the anti-repackaging measures that are in place. Out of the four apps used in our case studies, spanning both closed and open source Instant Messaging and SMS apps, one app was found to be using anti-repackaging measures. While many of these measures are well documented and easily bypassable, as was the case in this work's final case study, others may require app-specific knowledge and demand further reverse engineering efforts. Further experimentation is required to assess how popular these measures are across top apps in app stores, and if so, what percentage require app-specific knowledge. In the case that these measures are widely adopted and a majority require app-specific knowledge, *selective symbolic execution* (S^2E) [73] may be adopted to determine the app execution path values that result in success; i.e. what parameter values within the app codebase result in code integrity checks being passed. A selective approach would be required to avoid path explosion and would focus on the initial portion of the app code that is in charge of launching and setting up the app, given that these checks are typically made once upon installation.

All the case studies considered in this research required manual effort to determine successful trigger points and the evidence data objects of interest needed to develop JIT-MF drivers per incident and target app pair. The effort needed is lessened through black-box trigger points. However, a manual approach is still required uniquely per incident and target app pair. This does not scale with the ever-growing attack vectors, the sheer number of potential target apps, and the frequent changes in an app's codebase with each new version of an app. A possible solution could be creating more robust JIT-MF drivers that can cater to multiple app-attack scenarios, using a category of black-box trigger points related to infrastructure that is common to several apps, e.g. Firebase app messaging and SQLite functions. In this case, several incident and target app pairs can use the same trigger point, providing they share common underlying infrastructure functions. Manual effort is then only required to determine the evidence data object of interest (unless this is also obtained as a by-product of the chosen trigger points), object carving and parsing. Since data objects are specific to each app, parsing efforts are unique to each app-attack scenario, similar to other forensic tools like Plaso parsers.

Simultaneous dynamic instrumentation of multiple processes running within an app is another challenge that JIT-MF tools face. From all the case studies considered, those involving the Pushbullet app required a particular subprocess that was spawning to be instrumented. While in this case, selecting and instrumenting the class spawning a specific background subprocess was sufficient, other apps not considered in this study may require multiple processes to be instrumented simultaneously. The current implementation of MobFor relies on Frida Gadget, a DBI tool that leverages static library injection to avoid rooting the device. In MobFor's case, a Frida Gadget shared library containing instrumentation code is loaded inside the process memory. The setup required for Frida Gadget to hook into functions occurs when the library is loaded. If the Frida gadget library is loaded in the main app process, in that case, any subprocesses spawned from the main app process have a copy in memory of the libraries loaded by the main process and cannot reload the library. Therefore, when the library is already loaded in the main process of an app, hooking of function calls that occur in the subprocess cannot be performed. At the same time, if a function call occurs in the subprocess, any instrumentation targeting functions within the subprocess will not execute due to them being called from a different process than where the library was loaded. Therefore, MobFor can only instrument one of the processes spawned by an app can be instrumented at a time. That said, possible approaches that could be taken in the future include making contributions to the existing Frida implementation that enable the initialisation of the setup required for hooking through another method that does not require loading the Frida gadget library. Furthermore, the efforts made towards finding a solution need to be justified following a study on the popularity of apps spawning multiple background processes and the requirement of having them all instrumented simultaneously since the latter was not the case among the Pushbullet case studies considered.

Finally, while the JIT-MF framework is designed to work with any target app and investigative scenario pair, this does not currently include system apps. Unlike user-installed apps downloaded from app stores, system apps are typically specific to the device manufacturer. They come pre-installed in the system partition and cannot be uninstalled. Therefore, the initial forensic readiness stage required by JIT-MF tools cannot be carried out. While the number of system apps installed on a device is small, compared to user-installed apps, essential system apps may comprise default SMS apps that can be misused in messaging hijack attacks. In the case of investigative scenarios that include system app functionality, app-virtualisation frameworks may be used to allow an instrumented version of the system app, modified by MobFor, to run in parallel to the original version. The device owner would then need to ensure that the instrumented app is changed to the default.

IX. CONCLUSION

Android malware employing stealthy attacks retain a longer lifespan on the victim's device, often causing late detection and requiring incident response. Being stealthy in nature, such attacks leave a severely reduced set of forensic footprints in stored data. Evidence of such malware is only available as ephemeral data objects in volatile memory. Consequently, state-of-the-art forensic tools focusing on collecting evidence from stored sources on the device cannot construct comprehensive forensic timelines to aid investigators during an investigation.

In this work, we address this problem in scenarios where the device owner is a potential victim of a cybercrime attack related to messaging hijack attacks and motivate

the need for a memory forensics approach, as adopted by JIT-MF. Specifically, in this work, we sought to answer three research questions, whose aim was to: i) motivate the need for JIT-MF tools ii) address implementation challenges involved in creating such tools, and iii) demonstrate the values of such tools in a realistic setting.

Results show that JIT-MF does improve on timelines generated by baseline sources without compromising the security of the device, hence further motivating the need for JIT-MF tools. The challenges presented by JIT-MF tools are addressed by creating MobFor, a JIT-MF tool. Specifically, results show that trigger point selection can be successfully carried out through the adoption of black-box trigger points, which require minimal reverse engineering effort, and sampling can be used to considerably reduce any additional performance overheads incurred while still effectively producing forensic artefacts critical to an investigation. Finally, we demonstrate the unique way in which MobFor contributes to state-of-the-art forensic tools (XRY and Belkasoft) by providing critical missing forensic artefacts in a realistic setting involving the WhatsApp Pink malware and stock Android devices; the presence of which aids in generating a comprehensive forensic timeline including all the attack steps carried out. While successfully demonstrating how the JIT-MF framework can contribute in practice, this work raised several new research questions, mainly regarding privacy and automation of JIT-MF driver development in other attack scenarios, that call for further experimentation.

ACKNOWLEDGMENTS

This work is supported by the LOCARD Project under Grant H2020-SU-SEC-2018-832735.

REFERENCES

[1] Shanhong L. Android - statistics & facts. https://www.statista.com/topics/876/android/, 2020.
[2] Google play protect. https://www.android.com/play-protect/, 2020.
[3] Platon Kotzias, Juan Caballero, and Leyla 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.
[4] Y. Fratantonio, C. Qian, Simon 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.
[5] Marco Alecci, Riccardo Cestaro, Mauro Conti, Ketan Kanishka, and Eleonora Losiouk. Mascara: A novel attack leveraging android virtualization. arXiv preprint arXiv:2010.10639, 2020.
[6] Triada: organized crime on android. https://www.kaspersky.com/blog/triada-trojan/11481/, 2016.
[7] 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.
[8] Michalis Diamantaris, Elias P Papadopoulos, Evangelos P Markatos, Sotiris Ioannidis, and Jason Polakis. Reaper: real-time app analysis for augmenting the Android permission system. In ACM CODASPY, pages 37–48, 2019.
[9] 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.
[10] Stefanko, Lukas. Insidious Android malware gives up all malicious features but one to gain stealth. https://www.welivesecurity.com/2020/05/22/insidious-android-malware-gives-up-all-malicious-features-but-one-gain-stealth/, 2020. Accessed: 24.03.2021.
[11] ThreatFabric. 2020 - year of the RAT. https://www.threatfabric.com/blogs/2020_year_of_the_rat.html, 2020. Accessed: 24.03.2021.
[12] Flubot malware – all you need to know & to act now. https://www.threatmark.com/flubot-banking-malware/, 2021.
[13] Yonas Leguesse, Mark Vella, Christian Colombo, and Julio Hernandez-Castro. Reducing the forensic footprint with Android accessibility attacks. In STM, pages 22–38, 2020.
[14] M. Vella and V. Rudramurthy. Volatile memory-centric investigation of SMS-hijacked phones: a Pushbullet case study. In FedCSIS, pages 607–616. IEEE, 2018.
[15] Andrew Case and Golden G Richard III. Memory forensics: The path forward. Digital Investigation, 20:23–33, 2017.
[16] Andrew Case, Ryan D Maggio, Md Firoz-Ul-Amin, Mohammad M Jalalzai, Aisha Ali-Gombe, Mingxuan Sun, and Golden G Richard III. Hooktracer: Automatic detection and analysis of keystroke loggers using memory forensics. Computers & Security, 96:101872, 2020.
[17] Aisha Ali-Gombe, Alexandra Tambaoan, Angela Gurfolino, and Golden G Richard III. App-agnostic post-execution semantic analysis of Android in-memory forensics artifacts. In ACSAC, pages 28–41, 2020.
[18] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha I Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Tipped off by your memory allocator: Device-wide user activity sequencing from Android memory images. In NDSS, 2018.
[19] Benjamin Taubmann, Omar Alabduljaleel, and Hans P Reiser. DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps. Digital Investigation, 26:S67–S76, 2018.
[20] Aisha Ali-Gombe, Sneha Sudhakaran, Andrew Case, and Golden G Richard III. DroidScraper: a tool for Android in-memory object recovery and reconstruction. In $RAID$, pages 547–559, 2019.
[21] Jennifer Bellizzi, Mark Vella, Christian Colombo, and Julio Hernandez-Castro. Real-time triggering of Android memory dumps for stealthy attack investigation. In NordSec, pages 20–36, 2021.
[22] Jennifer Bellizzi, Mark Vella, Christian Colombo, and Julio Hernandez-Castro. 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 2021, July 6-8, 2021, pages 356–369. SCITEPRESS, 2021.
[23] Jason T. Luttgens, Matthew Pepe, and Kevin Mandia. Incident Response & Computer Forensics. McGraw-Hill Education Group, 3rd edition, 2014.
[24] Andrew Hoog. Android forensics: investigation, analysis and mobile security for Google Android. 2011.
[25] Jasmin Cosic and Miroslav Baca. A framework to (im) prove "chain of custody" in digital investigation process. In CECIIS, page 435, 2010.
[26] Gustuff: Weapon of mass infection. https://www.group-ib.com/blog/gustuff, 2019.
[27] Eventbot: A new mobile banking trojan is born. https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born, 2020.
[28] 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.
[29] 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.
[30] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui 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.
[31] Manar Mohamed, Babins Shrestha, and Nitesh Saxena. Smashed: Sniffing and manipulating android sensor data for offensive purposes. IEEE Transactions on Information Forensics and Security, 12(4):901–913, 2016.
[32] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. Forensic analysis of telegram messenger on android smartphones. Digital Investigation, 23:31–49, dec 2017.
[33] Google. File-based encryption. https://source.android.com/security/encryption/file-based Accessed: 24.03.2021.
[34] Himanshu Srivastava and Shashikala Tapaswi. Logical acquisition and analysis of data from android mobile devices. Information & Computer Security, 2015.
[35] Android developer's guide. https://developer.android.com/guide/topics/data/autobackup, 2020.
[36] Belkasoft. How to acquire data from an android device using apk downgrade method. https://belkasoft.com/Android_APK_downgrade_method. Accessed: 8.11.2021.

[37] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, and Taejoo Chang. New acquisition method based on firmware update protocols for android smartphones. Digital Investigation, 14:S68–S76, 2015.

[38] Belkasoft evidence centre x. https://belkasoft.com/x, 2021.

[39] Msab xry. https://www.msab.com/wp-content/uploads/2021/11/XRY_Logical_EN.pdf, 2021.

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

[41] Kristinn Guðjónsson. Mastering the super timeline with log2timeline. SANS Institute, 2010.

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

[43] Android developer's guide. https://developer.android.com/, 2020.

[44] M. Vitas. Art vs dalvik introducing the new android runtime in kitkat. https://infinum.com/the-capsized-eight/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat, 2020.

[45] T. Surin. Dealing with large memory requirements on android. https://pspdfkit.com/blog/2019/android-large-memory-requirements/, 2020.

[46] Volatility framework. https://www.volatilityfoundation.org/.

[47] Joe Sylve. Lime-linux memory extractor. In Proceedings of the 7th ShmooCon conference, 2012.

[48] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, Rohit Bhatia, Brendan Saltaformaggio, and Dongyan Xu. Live acquisition of main memory data from android smartphones and smartwatches. Digital Investigation, 23:50–62, 2017.

[49] Lief: Library to instrument executable formats. https://lief-project.github.io/, 2021.

[50] Understanding the xposed framework for art runtime. https://mssun.me/blog/understanding-xposed-framework-art-runtime.html, 2015.

[51] Geonbae Na, Jongsu Lim, Sunjun Lee, and Jeong Yi. Mobile code anti-reversing scheme based on bytecode trapping in art. Sensors, 19:2625, 06 2019.

[52] Dennis Andriesse. Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. no starch press, 2018.

[53] Frida. https://frida.re/, 2021.

[54] Alex Akinbi and Ehizojie Ojie. Forensic analysis of open-source XMPP/-Jabber multi-client instant messaging apps on Android smartphones. SN Applied Sciences, 3(4):1–14, 2021.

[55] G. Satrya, P. Daely, and M. Nugroho. Digital forensic analysis of telegram messenger on android devices. In ICTS, pages 1–7. IEEE, 2016.

[56] L. Zhang, F. Yu, and Q. Ji. The forensic analysis of WeChat message. In IMCCC, pages 500–503. IEEE, 2016.

[57] M. Azhar and T. Barton. Forensic analysis of secure ephemeral messaging applications on android platforms. In ICGS3, pages 27–41. Springer, 2017.

[58] Cosimo Anglano. Forensic analysis of WhatsApp messenger on Android smartphones. Digital Investigation, 11(3):201–213, 2014.

[59] Aditya Mahajan, MS Dahiya, and HP Sanghvi. Forensic analysis of instant messenger applications on Android devices. arXiv:1304.4915, 2013.

[60] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. A cloud-focused mobile forensics methodology. IEEE Cloud Computing, 2(4):60–65, 2015.

[61] Rekall Team. Rekall memory forensic framework: about the rekall memory forensic framework. Retrieved March, 13:2015, 2015.

[62] 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.

[63] 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. ACM SIGSAC, 2015.

[64] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In $USENIX$, pages 1137–1151, 2016.

[65] 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.

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

[67] 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.

[68] Yoan Chabot, Aurélie Bertaux, Christophe Nicolle, and M-Tahar Kechadi. A complete formalized knowledge representation model for advanced digital forensics timeline analysis. Digital Investigation, 11:S95–S105, 2014.

[69] 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.

[70] Rusydi Umar, Imam Riadi, and Guntur Maulana Zamroni. A comparative study of forensic tools for whatsapp analysis using nist measurements. Int. J. Adv. Comput. Sci. Appl, 8(12):69–75, 2017.

[71] Rusydi Umar, Imam Riadi, Guntur Maulana Zamroni, et al. Mobile forensic tools evaluation for digital crime investigation. Int. J. Adv. Sci. Eng. Inf. Technol, 8(3):949, 2018.

[72] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. Journal of Computer Security, 19(5):895–934, 2011.

[73] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), number CONF, 2009.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3160531, IEEE Access

JENNIFER BELLIZZI is a computer science PhD student at the University of Malta and currently the lead researcher on the EU-funded H2020 Project "LOCARD" in the Computer Science Department at the University of Malta. She received her B.Sc. in Computer Science and Artificial Intelligence from the University of Malta in 2014 and pursued her studies at the University of Birmingham in the UK, where she received her M.Sc. in Cyber Security in 2016. She has worked in cybersecurity across various sectors, including LEAs, the private sector, and academia. Her interest in incident response and digital forensics stems from her time working in the private sector, where she was responsible for implementing incident response strategies and investigating potential threats. Her research interests include malware analysis, incident response, and memory forensics.

MARK VELLA currently holds the position of Senior Lecturer at the University of Malta. He obtained an M.Sc in Computer Science from the University of Malta and spent a number of years participating and leading enterprise application and integration projects before moving back to academia. He pursued a research doctorate in the area of computer systems security at the University of Strathclyde (UK). His initial research on developing intrusion detection techniques inspired by the workings of the human immune system, has today found home and immediate application within the context of using memory forensics for incident response, while keeping an eye on making computer systems less prone to security breaches. At university he lectures and advises undergraduate and postgraduate students on topics of computer systems and security.

CHRISTIAN COLOMBO received his B.Sc., M.Sc., and Ph.D. degrees in Computer Science from the University of Malta in 2007, 2009, and2013 respectively. From 2008 to 2010, he worked as a research assistant on the nationally-funded project, Dependability and Error-Recovery in Security Intensive Financial Systems. Since 2010, he has been employed as an academic within the Department of Computer Science at the University of Malta. His research interests include runtime verification, software testing, compensating transactions, and domain-specific languages, with over 50 publications in these areas. He is currently focused on applying runtime verification in the area of cyber security through the funded projects: Secure Communication in the Quantum Era (NATO) and Lawful Evidence Collecting & Continuity Platform Development (Horizon2020). Dr. Christian Colombo was a recipient of the MGSS Scholarship Scheme 2008

JULIO HERNANDEZ-CASTRO was with the University of Portsmouth,U.K., and Carlos III University, Spain. He is also affiliated with the Kent Cybersecurity Center. He is currently a Professor of computer security with the School of Computing, University of Kent. His research interests are wide, covering from RFID security to lightweight cryptography, including steganography and steganalysis and the design and analysis of CAPTCHAs. He has been a Pre-Doctoral Marie Curie Fellow and also a Post-Doctoral INRIA Fellow. He is currently the Vice-Chair of the EU COST Project CRYPTACUS. He receives research funding from Innovate UK Project aS, EPSRC Project 13375, andEU H2020 Project RAMSES.