Responding to Living-Off-the-Land Tactics using Just-In-Time Memory Forensics (JIT-MF) for Android

Jennifer Bellizzi[1][a], Mark Vella[1][b], Christian Colombo[1][c] and Julio Hernandez-Castro[2][d]

[1]*Department of Computer Science, University of Malta, Msida, Malta*

[2]*School of Computing, Cornwallis South, University of Kent, Canterbury, U.K.*

christian.colombo}@um.edu.mt,

Keywords: Memory Forensics, Android Security, Digital Forensics, Incident Response, Forensic Timelines.

Abstract: Digital investigations of stealthy attacks on Android devices pose particular challenges to incident responders. Whereas consequential late detection demands accurate and comprehensive forensic timelines to reconstruct all malicious activities, reduced forensic footprints with minimal malware involvement, such as when Living-Off-the-Land (LOtL) tactics are adopted, leave investigators little evidence to work with. Volatile memory forensics can be an effective approach since app execution of any form is always bound to leave a trail of evidence in memory, even if perhaps ephemeral. Just-In-Time Memory Forensics (JIT-MF) is a recently proposed technique that describes a framework to process memory forensics on existing stock Android devices, without compromising their security by requiring them to be rooted. Within this framework, JIT-MF drivers are designed to promptly dump in-memory evidence related to app usage or misuse. In this work, we primarily introduce a conceptualized presentation of JIT-MF drivers. Subsequently, through a series of case studies involving the hijacking of widely-used messaging apps, we show that when the target apps are forensically enhanced with JIT-MF drivers, investigators can generate richer forensic timelines to support their investigation, which are on average 26% closer to ground truth.

1 INTRODUCTION

The use of process memory forensics is increasingly becoming a necessity when investigating advanced stealthy cyberattack incidents targeting smartphones (Case et al., 2020; Ali-Gombe et al., 2020; Bhatia et al., 2018; Taubmann et al., 2018). This is the reality incident responders face amidst the limitations and barriers of the more forensically sound, but alas limited, state-of-the-art mobile forensics. In this work, our primary concern is Android malware that exhibits long-term stealth by evading early detection mechanisms, which eventually is only detected through its consequences. Specifically, we focus on attacks that hijack the messaging functionality of Android devices to hide compromising communication of a criminal nature behind victim devices or else spy on them through unlawful interception. In the eventuality that device owners notice suspicious activity and hand their device over for further investigation, incident responders would need to investigate — utilizing a forensic timeline (Guðjónsson, 2010) — all the activities undertaken by threat actors through the deployed malware.

Android accessibility trojans are a case in point (Stefanko, Lukas, 2020; ThreatFabric, 2020). This attack vector has been shown to enable stealthy Living-Off-the-Land (LOtL) tactics (Campbell, Christopher and Graeber, Matthew, 2013), where key attack steps get delegated to benign apps, possibly only requiring the use of malware during an initial setup phase. In similar scenarios, trying to establish a forensic timeline solely from sources found on non-volatile flash memory, whether on-chip or removable, can prove futile even after many barriers to forensic evidence collection are overcome. The root cause is that the forensic artefacts constituting attack evidence would have been erased from storage or never even created to begin with. Volatile memory can be an effective forensic source in such circumstances. No matter how stealthy an attack can be, its execution through malware or benign victim apps has to occur in memory (Case and Richard III, 2017). Therefore, any result-

[a] https://orcid.org/0000-0003-1754-9473
[b] https://orcid.org/0000-0002-6483-9054
[c] https://orcid.org/0000-0002-2844-5728
[d] https://orcid.org/0000-0002-6432-5328

Bellizzi, J., Vella, M., Colombo, C. and Hernandez-Castro, J.
Responding to Living-Off-the-Land Tactics using Just-In-Time Memory Forensics (JIT-MF) for Android.
DOI: 10.5220/0010603603560369
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 356-369
ISBN: 978-989-758-524-1

ing evidence has to be present in process memory, even if just briefly. While full-device static dumps of volatile memory typically require an unlocked bootloader and customized firmware, dynamic dumps of process memory do not necessarily face similar restrictions (Bellizzi et al., 2021). We show that this is the case when deployed using a mix of static and dynamic app instrumentation, at least when not involving system apps and services.

Just-In-time Memory Forensics (JIT-MF) is a framework for live process memory forensics, with an initial study describing how to formulate and implement JIT-MF drivers (Bellizzi et al., 2021). These drivers are responsible for establishing the points in time when memory dumps should be triggered and the heap/native memory areas/objects to be included. While in this paper we implement drivers in the context of message hijacking, these are meant to be configurable to cater for other investigative scenarios unrelated to message hijacking.

Figure 1 shows the complete JIT-MF workflow. Starting with a forensic readiness stage (Luttgens et al., 2014), 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. Once a risk assessment is carried out, those apps that pose a particular risk, say messaging apps, are instrumented with JIT-MF drivers (step 2). With JIT-MF, the forensic acquisition of memory dumps is triggered by specific app events identified as trigger points by JIT-MF drivers (step 3). These memory dumps contain either raw binaries from memory segments or else readily-carved and parsed objects along with tagged metadata, e.g. in JSON format. These two acquisition methods are referred to as offline and online respectively, depending on how object carving and parsing are carried out.

Once suspicious activity is noticed, with alerts possibly raised by the device owners themselves or by incident responders during routine checks, the JIT-MF dumps are merged with other forensic sources to produce a forensic timeline (steps 4 and 5). JIT-MF's scope is that of producing a richer forensic timeline, aiming for a comprehensive reconstruction of app activity, supporting investigators to establish the full picture of the incident. Techniques, such as comparisons with a baseline of known normal device/app usage or cross-checking with cyber threat intelligence feeds, can then be used to identify attack-related, non-consented app usage. The entire workflow steps should be recorded, accompanied with full metadata, hashes, and digital signatures, thus keeping a complete chain of custody (Cosic and Baca, 2010). In this manner, all the acquired evidence remains admissible to a court of law if need be.

JIT-MF drivers can be deployed on stock non-rooted devices without requiring any firmware alterations and are compatible with encrypted devices. By avoiding an approach based entirely on forensically enhancing the Android kernel or individual app codebases, JIT-MF can work with what is already deployed in the field. This approach avoids making any form of imposition on future releases of Android and/or apps developed for the platform. While not clashing with device access controls, such as screen and bootloader locks, JIT-MF does require the device owner's collaboration. This assumption's realism rests on the fact that device owners are the potential victims rather than perpetrators.

The key proposition of this paper is that through JIT-MF tools and their drivers, timely captured volatile memory can be used as an additional forensic source to help reconstruct incident scenarios in a more comprehensive manner, better supporting incident investigations that involve stealthy, long-running cyberattacks targeting Android smartphones, and their owners alike. We make the following contributions:

- Provide a conceptual, generic description of JIT-MF drivers, along with a methodology for their implementations.
- Six JIT-MF drivers for popular messaging apps, covering both SMS and instant messaging, supporting investigators to determine any malicious/non-consented activity.
- Experimentation involving stealthy messaging hijack case studies, demonstrating how using JIT-MF drivers results in richer forensic timelines, as compared to solely relying on forensic sources used by state-of-the-art mobile forensics tools.

2 BACKGROUND

Given the nature of apps that are typically installed on smartphones, ranging from messaging, voice/video calls, to the camera, navigation, calendar, and social networking, just to name a few, mobile devices are nowadays both a rich source of evidence as well as a primary target for cyberattacks (Scrivens and Lin, 2017).

2.1 Android Forensics

Forensic Sources. Android on-chip and removable flash memory constitute primary forensic sources,

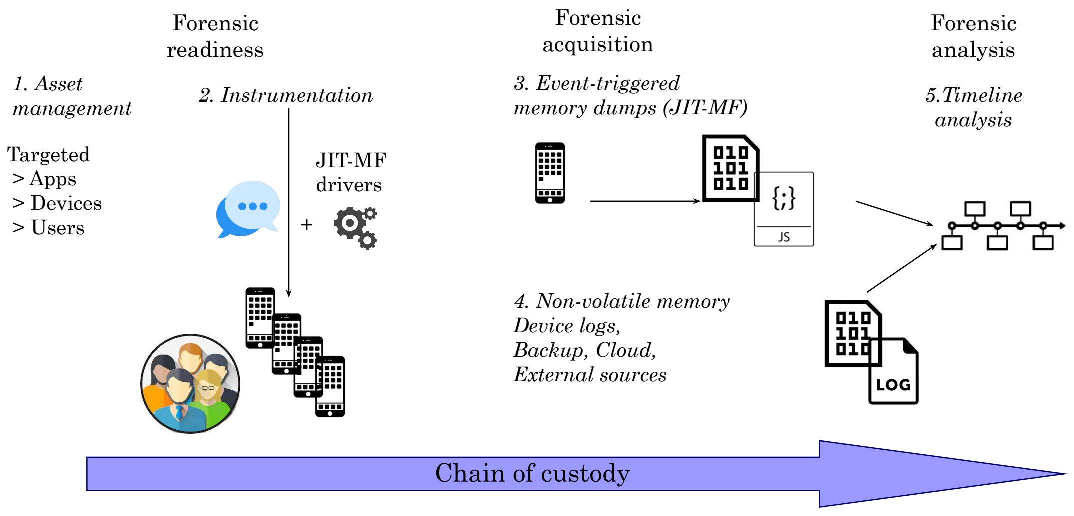


Figure 1: The JIT-MF workflow.

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`) (Hoog, 2011). The `/data/data` sub-tree of the Android file-system inside the `userdata` partition, along with the `sdcard` partition, 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 the available storage, 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 (Anglano et al., 2017)). Beyond the app-level, device-wide disk encryption has seen an evolution across Android versions. Full disk encryption (FDE) has now been replaced by file-based encryption (FBE) in Android 10 (Google, a), rendering it more practical and more stable, e.g. the alarm clock works even if the screen is locked and a full factory reset is no longer necessary if the device runs out of power before it shuts down properly.

Acquisition and Analysis. Evidence acquisition on Android can be carried out using logical or physical imaging (Srivastava and Tapaswi, 2015). Simply put, logical acquisition relies on some existing source to parse raw data to decode OS filesystem or app content. Android's filesystem and SQLite parsers are typical examples. App backup utilities and cloud storage interfaces are further examples. 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 acquisition methods have to deal with Android's security barriers. For software-based acquisition, the barriers range from locked screens to password-protected cloud storage and rooting the device to gain access to `/data/data/`. Rooting relies upon exploiting some kernel or firmware flashing protocol vulnerability (Yang et al., 2015; Srivastava and Tapaswi, 2015), or else flashing a custom recovery partition through which to add some root-privileged utility. The latter may get further complicated by locked bootloaders. While hardware-based acquisitions can bypass the above barriers, any form of physical imaging has to deal with FDE and FBE.

The starting point for forensic analysis pretty much depends on what kind of acquisition is performed (Hoog, 2011). In this case of physical acquisition, it is necessary to first identify the filesystem 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 onwards, decoding of app evidence is pretty much app-specific.

Mobile Forensics Tools. As such, any mobile forensics tool, e.g. Oxygen Forensics, or Cellebrite's UFED, can be seen as a collection of acquisition options, equipped with rooting exploits and hardware interfacing cables, passcode brute-forcing methods, along with parsing/analysis modules for filesystems, database, and app data formats. Ancillary analysis features, including timeline generators, can provide a final professional touch to the product.

2.2 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 hypothesized incident/crime scenario (Hargreaves and Patterson, 2012). The richer the timelines, the greater the support for an investigator to reconstruct an intrusion/crime scene, thereby answering critical questions about an incident.

In the case of a messaging hijack, see Figure 2 (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 2 (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.

2.3 Just-In-Time Memory Forensics (JIT-MF)

JIT-MF is a framework that enables live process memory forensics in a setting involving attacks that delegate key steps to benign apps, possibly only minimally employing malware. JIT-MF is conceived to be adopted by incident response tools for stock smartphones without breaking any of their security controls. Rather, given that its main purpose is to protect the device owner from cyberattacks and the perpetrators behind them, it assumes the device owner's collaboration. Initial experimentation showed JIT-MF's effectiveness to dump ephemeral artefacts at practical overheads. Moreover, while JIT-MF drivers are app- and incident scenario-specific by design, the work involved in defining trigger points only requires a black-box analysis of target apps. An in-depth comprehension only concerns the evidence objects themselves, e.g. the structure of objects representing message objects or messaging activity in SMS or instant messengers. On the downside, pending some form of privacy-preserving computation to be added to JIT-MF, privacy concerns must be addressed solely through procedural rather than technical controls. Limitations also apply in terms of clashes with anti-repackaging measures and dependency on Android RunTime (ART) data structures bound to change between Android versions.

2.4 Related Work

The topic of Android messaging timelining has already received attention from various works (Akinbi and Ojie, 2021; Anglano et al., 2017; Anglano, 2014). Their focus, however, is exclusive to content stored on internal flash memory, typically SQLite database schemas and any corresponding encryption and decryption key location. Also, the automated analysis of forensic timelines (Chabot et al., 2015; Mohammad and Alqahtani, 2019) is orthogonal to our work. JIT-MF's scope is to provide a solution for rendering the acquisition of incident-related evidence as comprehensive as possible, as otherwise no manual or automated process would be able to reconstruct the incident.

JIT-MF relies on a combination of static and dynamic instrumentation of compiled code to implement trigger points as in-line function hooks and the memory dumping process in the form of instrumentation code. Binary instrumentation also underpins various other Android security techniques where it is required to operate within real-time parameters (Diamantaris et al., 2019; Heuser et al., 2014; Chen et al., 2017; Li et al., 2015). Furthermore, JIT-MF considers the temporal aspect of volatile memory forensic collection concerned with the timely collection of ephemeral artifacts. Other temporal aspects that got the attention of similar memory forensics studies concern the memory smearing problem of full device memory dumps (Pagani et al., 2019) and the extraction of the temporal dimension from static dumps (Saltaformaggio et al., 2015; Ali-Gombe et al., 2019).

In this work, we focus on the attack vector presented by Android accessibility services since it presents an ongoing threat, with multiple recent incidents gaining a world-wide reach (Whittaker, Zack, 2020; Stefanko, Lukas, 2020; ThreatFabric, 2020). Crucially, for incident response, the resulting reduced forensic footprint for any attack employing it has also been demonstrated (Leguesse et al., 2020). Yet, other attack vectors may also present the same opportunity for LOtL tactics. Zygote and binder infection combined with a rooting exploit (Kaspersky, 2016), as well as app-level virtualization frameworks (Shi et al., 2019) and third-party library infections (Diamantaris et al., 2019) provide further attack vectors, resulting in similarly stealthy attacks and for which JIT-MF could be a solution in terms of incident response.

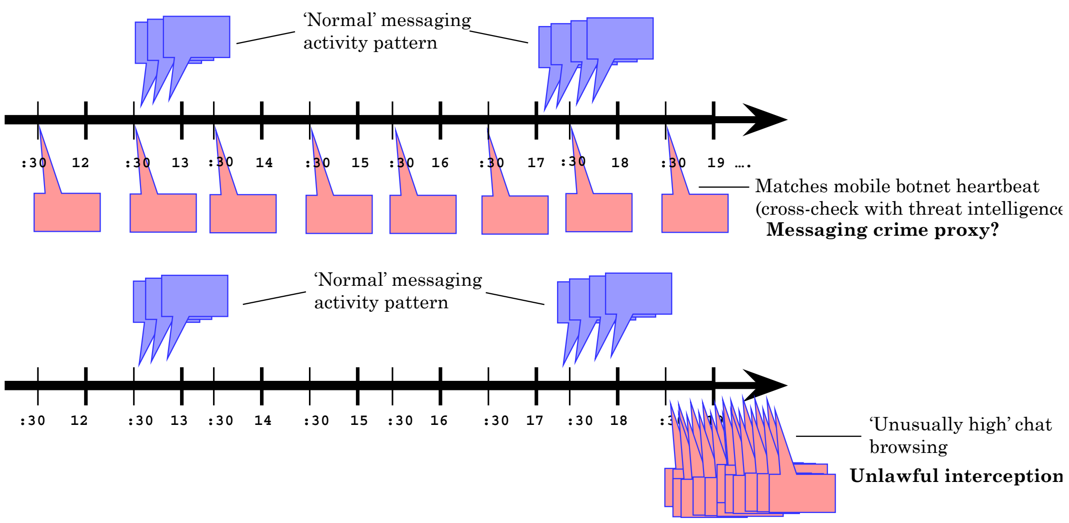

'Normal' messaging activity pattern
:30 12 :30 13 :30 14 :30 15 :30 16 :30 17 :30 18 :30 19 ....
Matches mobile botnet heartbeat (cross-check with threat intelligence
Messaging crime proxy?
'Normal' messaging activity pattern
:30 12 :30 13 :30 14 :30 15 :30 16 :30 17 :30 18
'Unusually high' chat browsing
Unlawful interception

Figure 2: Forensic timelines supporting cyber-attack investigations.

3 JIT-MF DRIVERS

While JIT-MF (Bellizzi et al., 2021) defines common steps to be followed by every JIT-MF tool, those aspects that are specific to the investigation scenario/target app pair at hand are described and eventually implemented by JIT-MF drivers. Conceptually, their definition starts with identifying the in-memory evidence 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. Trigger points can be rendered more specific through complementary conditions defined over function arguments. The acquisition method must be defined as either online or offline, specifying whether object carving/parsing is carried out prior to memory dumping or else afterwards. Finally, a sampling strategy is defined to maintain a manageable amount of dumps. Listing 1 presents a template generically describing JIT-MF drivers.

```
1   Driver_ID: string
2   Scope: <app, incident_scenario>
3   /*Attributes*/
4   Evidence_objects: {<event: string, object_name: string,
5       carve_object_type(), parse_object_type(), {
            trigger ids}>, ...}
6   Acquisition: {online | offline}
7   Triggers: {<trigger_id: string,<hooked_function_name:
          string,
8       level: {native|rt}, trigger_predicate()>,
9       trigger_callback() >,....}
10  Sampling_strategy: sampling_predicate()
11  Log_location: string
12  Globals: {<key,value>, ...}
13  /*Exposed interface*/
14  bool init(config: {<key,value>, ...}) {
15      set_globals(config: {<key,value>, ...});
16      for all entries in Triggers:
17          place_native_hook() ⊕ place_rt_hook();
18  }
19  /*Internal functions*/
20  bool trigger_predicate_i(params: {<key,value>, ...}) {
21      decide on whether to fire the corresponding
            trigger;
22  }
23  void trigger_callback_i(thread_context: {<key,value>,
        ...}) {
24      if trigger_predicate_i && sampling_predicate return true:
25          perform memory forensics on the current app
                state;
26  }
27  [object: address,...] carve_object_type_j(from: address,
28      to: address) {
29      attempt object carving in the given memory
            range;
30  }
31  [field: type,...] parse_object_type_j(at: address) {
32      parse object fields starting at the given
            address;
33  }
34  bool sampling_predicate(thread_context: {<key,value>,
        ...}) {
35      decide on whether to follow up a trigger by a
            memory dump;
36  }
```

Listing 1: JIT-MF driver template.

Lines 1-2 identify the driver (*Driver_ID*) and link it with its intended app/incident *Scope*. Lines 4-12 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 20-36). *Globals* is a key-value meant for miscellaneous usage.

init() presents the only interfaces exposed to the JIT-MF tool environment. It is called during tool initialization and takes care of initializing *Globals* and most importantly 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 both defined per entry in *Evidence_objects*, at least for online *Acquisition*. All these functions require a JIT-MF runtime for their implementation. Listing 2 presents this runtime assumed by JIT-MF drivers which needs to be catered for by the JIT-MF tool.

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

Listing 2: JIT-MF driver runtime.

Lines 1-3 are *Globals* access and *native*/*rt* function-hooking functions called from *init()* and any other driver internal functions as needed. Lines 4-16 are process memory interacting functions, starting off *list_memory_segments()* in order 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 *Acquisition*, calling *dump_memory_segment* may suffice, for online acquisition 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, taking the appropriate *carve_object_type()* and *parse_object_type()* callback functions as parameters. Separate *rt* and *native* versions are needed since the *rt* version may leverage calling runtime functions in order 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 17) is responsible to produce the actual JIT-MF dump to the location specified by the driver's *Log_location*.

4 METHODOLOGY

Before proceeding to demonstrate the value that JIT-MF drivers (section 5) add through their role as forensic sources in incident response, we first explain how specific JIT-MF drivers were constructed for the case studies considered. We also describe the methodology used to merge the JIT-MF forensic sources with all other available sources and the eventual resulting timelines of the incidents concerned.

4.1 JIT-MF Driver Definition

Taking previous experiment results (Bellizzi et al., 2021) as guidance for JIT-MF driver definition, white-box analysis of the apps concerned should be restricted to *Evidence_objects* identification and implementing their corresponding carving/parsing. Consequently, some form of app code comprehension or reversing is required. On the other hand, *Triggers* identification should follow a black-box approach. The implication is that the hooked functions can be identified based on calls made to Android APIs, well-known third-party libraries, or even Linux system calls performed by the app, rather than hooking the app's internal functions. The finalized methodology that was adopted in this regard 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 evidence objects, trigger points are selected based on the code that processes the said objects, specifically concerning: *i)* The storing and loading of the objects from storage; *ii)* The transferring of objects over the network; or else *iii)* Any object transformation of some sort (e.g. display on-screen etc.). As yet, we are limited to introducing trigger points only in the *main process* of an app.

4.2 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. Whenever the same data could be obtained from multiple sources, we opt for local device acquisition, rather than cloud or backups, to facilitate experimentation. These forensic sources, as explained in section 2.1, 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 acquisition and content decryption. These baseline sources are complemented by those provided by JIT-MF drivers.

Figure 3 shows the processes that transform the forensic footprints obtained from the aforementioned forensic sources to finished forensic timelines. This pipeline is based on Chabot et al.'s (Chabot et al., 2014) methodology. It revolves around the creation of a knowledge representation model as derived from multiple forensic sources. It presents a canonical semantic view of the combined sources upon which forensic timeline (or other) analysis can be conducted. This model is populated with scenario events derived from forensic footprints, 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. Relations established by this process correspond either to a relation of composition or causation.

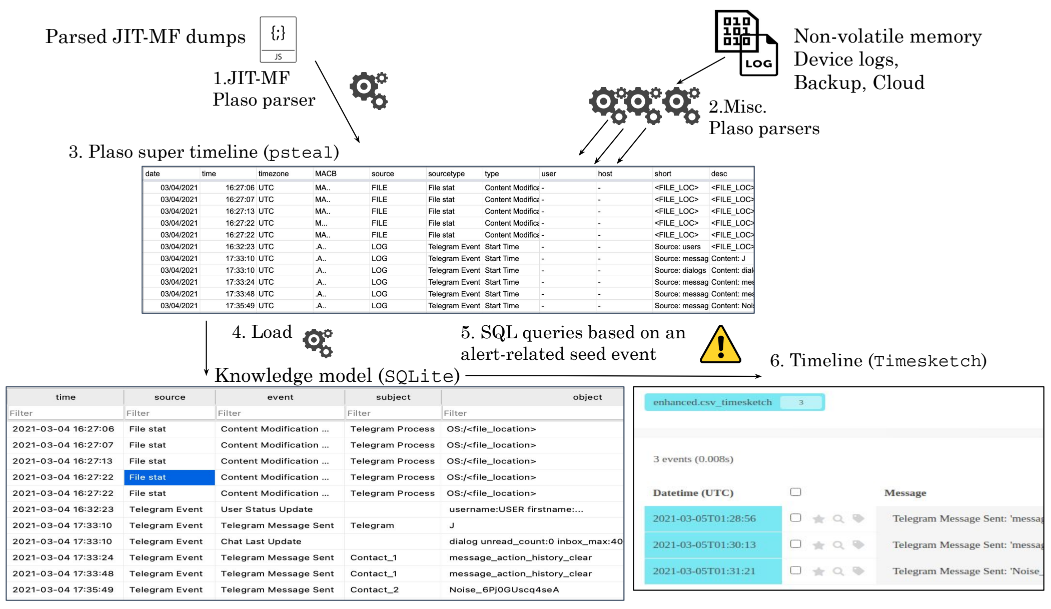
The first three steps in Figure 3 consist of forensic footprint extraction. For JIT-MF we refer to a combined dump containing unique, readily carved and parsed memory objects. All sources are decoded and merged as a Plaso (Guðjónsson, 2010) super timeline using the `psteal` utility, and for which we developed a JIT-MF Plaso parser. 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 event is a message 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 based on a 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` (Google, b) for visualization.

5 EXPERIMENTATION

To demonstrate the added value that JIT-MF tools provide during incident response, in comparison with current mobile forensics tools, 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 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 unlawful interception 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, a rooted emulator is used 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[1] runtime, and JIT-MF drivers were implemented as Javascript code for Frida's Gadget shared library. The JIT-MF drivers used in all case studies have the following attributes: *Acquisition = online*, *Sampling = 1 in 5* (active for a second every 5 seconds) and

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


Parsed JIT-MF dumps
JS
1.JIT-MF Plaso parser
Non-volatile memory Device logs, Backup, Cloud
LOG
2.Misc. Plaso parsers
3. Plaso super timeline (psteal)
4. Load
5. SQL queries based on an alert-related seed event
Knowledge model (SQLite)
6. Timeline (Timesketch)
enhanced.csv_timesketch 3
3 events (0.008s)

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

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

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

Figure 3: The forensic timeline generation processes.

Log_location=/<external_storage>/jitmflogs. Table 1 lists the properties of the state-of-the-art forensic sources considered, their method of acquisition and required parsers for populating a Plaso super timeline. Apart from the JIT-MF Plaso parser, additional parsers for the app-specific databases (including write-ahead-log database) and `logcat` were created.

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, 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 off from a seed event. In each case study, the malicious event (attack) is executed three times to ensure that the ground truth timeline contains more than one event. Noise is generated with respect to this value. Some of the attacks used in these case studies make use of rate-limited API calls to a server back-end, which only allows 150 consecutive calls to be made from the same device. Since an attack is executed three times per case study, and the API call limit is 150, each case study is simulated 50 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 either of 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.

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

Table 1: Forensic Sources and Parsers.

Case Study	Location on device	Source type	Contents	Acquisition & 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

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 10 - 20 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 10 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 that has the below attributes:

Evidence_objects: {<"Telegram Message Sent", "`org.telegram.messenger.MessageObject`", {"TG_CP}">}

Triggers: {<"TG_CP<", "`send`", native, network≫}

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.

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 that has the below attributes:

Evidence_objects: {<"Signal Message Sent","`org.thoughtcrime.securesms.conversation.ConversationMessage`", {"SIG_CP"}>}

Triggers: {<"SIGNAL_CP",<"`write`", native, storage≫}

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.

C: Pushbullet Crime-Proxy

Accessibility Attack. John's Facebook credentials are stolen by an attacker using a phishing accessibility attack akin to Eventbot(Whittaker, Zack, 2020). 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 10 - 20 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 that has the below attributes:
Evidence_objects: {<"Message Sent","`org.json.JSONObject`",{"PB_CP"}>}
Triggers: {<"PB_CP",<"`write`",native,storage≫}
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.
Seed Event: *<Subject: *suspicious number*, Event type: Message Sent>, Time: last month*
Matching Criteria: A message sent to the suspicious number.

D: Telegram Unlawful Interception

Accessibility Attack. An accessibility attack targets John's Telegram app and is used by an attacker to intercept messages sent to the username "CEO" (John's boss - with whom confidential data is shared). The attacker misuses John's Telegram app to grab messages exchanged with "CEO" and Telegram.
Setup. John 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 10 - 20 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 that has the below attributes:
Evidence_objects: {<"Telegram Chat Intercepted", "`org.telegram.messenger.MessageObject`", {"TG_SP"}>}
Triggers: {<"TG_SP",<"`recv`",native,network≫}
In the case of an unlawful interception attack, 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.

E: Signal Unlawful Interception

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 that has the below attributes:
Evidence_objects: {<"Signal Chat Intercepted", "`org.thoughtcrime.securesms.conversation.ConversationMessage`",{"SIGNAL_SP"}>}
Triggers: {<"SIG_SP",<"`open`",native,storage≫}
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.

F: Pushbullet Unlawful Interception

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 that has the below attributes:
Evidence_objects: {<"Chat Intercepted", "`org.json.JSONObject`",{"PB_SP"}}
Triggers: {<"PB_SP",<"`android.content.Intent.createFromParcel`",rt,network≫}
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 C5 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.
Seed Event: *Subject: CEO, Object: *confidential message*, Event type: Message Read/Loaded/Chat activity, Time: date of message sent/received*
Matching Criteria: As previous case study.

5.1 Results

Table 2 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 of 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 that were captured. Timeline difference was calculated using *Jaccard dissimilarity* on the set of true events uncovered by the generated timelines and the ground truth timeline. In this case, the higher the value between 0 and 1, the more dissimilar 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*, 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 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 unlawful interception (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 sub-processes that were not instrumented, and ii) The evidence object defined in the JIT-MF driver was a coarser-grained object (JSON object containing "push" event). Both of these limitations in JIT-MF's driver implementation contributed to a JIT-MF timeline whose gain on the baseline timeline was minimal, with regard to ground truth timeline similarity.

Table 2: Timeline comparison.

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

Furthermore, while JIT-MF timelines are more similar to the ground truth timeline than baseline timelines in cases involving unlawful interception (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 unlawful interception scenarios, we are after coarser-grained events (an indication of a chat being intercepted/synced) since key objects in memory are either not present or do not contain indicative metadata of the ongoing event.

JIT-MF Timeline Sequence Accuracy. Using *Kendall Tau distance*, 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. Additionally, the standard deviation between the time of the events logged in the ground truth timelines and that logged in JIT-MF timelines deviates on average by at most 62s.

Performance Overheads. The practical overheads recorded in the initial JIT-MF study (Bellizzi et al., 2021) were confirmed. 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. Janky frames[2] is an indicator of non-smooth user interactions with GUI apps. With JIT-MF drivers enabled Telegram and Signal had an average increase of 1.59% and 3.53% of Janky frames, respectively; that is, the performance penalty overall was less than 4%.

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

6 DISCUSSION

Provision of Context for Evidence Objects by Trigger Points. The selection of trigger points is not only crucial to solving the problem of timely dumping ephemeral evidence objects in memory. Trigger points also provide the necessary context for a dumped evidence object. Results show that JIT-MF tools are most successful in generating a quasi-identical timeline to the ground truth timeline when both trigger point *and* the evidence object are tightly linked to the attack step that needs to be uncovered. For instance, a `send` system call and a message object are directly linked to an attack step involving an attacker misusing a victim's phone to proxy a message.

Level of Event Granularity Associated with In-memory Objects. The level of granularity for the events associated with in-memory objects is not always enough for accurate association with specific app use/misuse events. In the case of coarser-grained events, due to the lack of metadata within the collected evidence, we noticed that, while still present, the additional information as compared to baseline timelines diminishes considerably. This seems to be the root cause for a diminishing return when we compared crime-proxy timelines with those obtained for unlawful interception.

Further Evolution of JIT-MF. JIT-MF operates under the assumption that by targeting evidence objects representing core app functionality (e.g. message objects in messaging apps), we may uncover relevant attack events. However, the issues mentioned above emerge as a result of this. An approach that merits further investigation is to produce a JIT-MF driver such that: i) Selected trigger points could provide more context to the evidence object being dumped and ii) Evidence objects can be more tightly coupled with app use/misuse. This would mean that although a black-box approach to selecting trigger points is successful in dumping ephemeral objects in memory, a white-box approach might be preferred, particularly in scenarios where the attack step does not have fine-grained metadata related to ongoing activity, necessary to deduce ongoing actions. Therefore the evidence object itself requires further context. Deeper app analysis may also be required when selecting an evidence object more suited to target specific attack steps, which becomes even more challenging when addressing obfuscated code where the implementation of an object within the app is obscure.

Threats to Validity. Except for Pushbullet, the two forensic sources making up the baseline in four of the six case studies are `logcat` and write-ahead log files containing incomplete forensic footprints and are subject to frequent log rotation. When performing longer runs with more iterations of the attack component, we noticed that out of 250 deleted messages, the write-ahead log file could only expose 20 unique messages. The same argument can be made for `logcat`. Since these logs are not captured in a timely and permanent manner, they would be futile in the case of a long-term stealthy attack. The same point cannot be said for Pushbullet based on the obtained results. However, a slightly more sophisticated malware consisting of a second stage which deletes sent messages on the device would expose the limitations of the existing baseline. On the other hand, a JIT-MF-based tool can timely capture and store the evidence object for the possible forensic analysis required at a later stage. A sampling property enables the tool to reduce the amount of storage occupied. However, if the device is running low on storage, then a JIT-MF based tool would run into the same issues of evidence availability as the baseline sources.

Finally, 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.

7 CONCLUSIONS

Stealthy Android malware based on accessibility has made it difficult for digital forensic investigators to uncover the attack steps involved in an incident. Previous work has shown that using a JIT-MF tool is crucial in dumping key ephemeral in-memory objects that can be linked to attack steps. In this paper, we demonstrate the extent to which JIT-MF tools can enrich forensic sources by comparing the enhanced timelines generated by JIT-MF tools to timelines generated by existing baseline sources. Results show that while JIT-MF does improve on timelines generated by baseline sources without compromising the security of the device, the information gain obtained in the timeline depends on the level of granularity of the attack step linked to the identified evidence object.

ACKNOWLEDGEMENTS

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

REFERENCES

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

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

Ali-Gombe, A., Tambaoan, A., Gurfolino, A., and Richard III, G. G. (2020). App-agnostic post-execution semantic analysis of Android in-memory forensics artifacts. In *ACSAC*, pages 28–41.

Anglano, C. (2014). Forensic analysis of WhatsApp messenger on Android smartphones. *Digital Investigation*, 11(3):201–213.

Anglano, C., Canonico, M., and Guazzone, M. (2017). Forensic analysis of telegram messenger on android smartphones. *Digital Investigation*, 23:31–49.

Bellizzi, J., Vella, M., Colombo, C., and Hernandez-Castro, J. (2021). Real-time triggering of Android memory dumps for stealthy attack investigation. In *NordSec*, pages 20–36.

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

Campbell, Christopher and Graeber, Matthew (2013). Living Off the Land: A Minimalist's Guide to Windows

Post-Exploitation. http://www.irongeek.com. Accessed: 24.03.2021.

Case, A., Maggio, R. D., Firoz-Ul-Amin, M., Jalalzai, M. M., Ali-Gombe, A., Sun, M., and Richard III, G. G. (2020). Hooktracer: Automatic detection and analysis of keystroke loggers using memory forensics. *Computers & Security*, 96:101872.

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

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

Chabot, Y., Bertaux, A., Nicolle, C., and Kechadi, T. (2015). An ontology-based approach for the reconstruction and analysis of digital incidents timelines. *Digital Investigation*, 15:83–100.

Chen, J., Wang, C., Zhao, Z., Chen, K., Du, R., and Ahn, G.-J. (2017). Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE TIFS*, 13(5):1286–1300.

Cosic, J. and Baca, M. (2010). A framework to (im) prove "chain of custody" in digital investigation process. In *CECIIS*, page 435.

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

Google. File-based encryption. https://source.android.com/security/encryption/file-based Accessed: 24.03.2021.

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

Guðjónsson, K. (2010). Mastering the super timeline with log2timeline. *SANS Institute*.

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

Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. (2014). ASM: A programmable interface for extending Android security. In *USENIX*, pages 1005–1019.

Hoog, A. (2011). *Android forensics: investigation, analysis and mobile security for Google Android*.

Kaspersky (2016). Triada: organized crime on Android. https://www.kaspersky.com/blog/triada-trojan/11481. Accessed: 24.03.2021.

Leguesse, Y., Vella, M., Colombo, C., and Hernandez-Castro, J. (2020). Reducing the forensic footprint with Android accessibility attacks. In *STM*, pages 22–38.

Li, S., Chen, J., Spyridopoulos, T., Andriotis, P., Ludwiniak, R., and Russell, G. (2015). Real-time monitoring of privacy abuses and intrusion detection in Android system. In *HAS*, pages 379–390.

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

Mohammad, R. M. A. and Alqahtani, M. (2019). A comparison of machine learning techniques for file system forensics analysis. *Journal of Information Security and Applications*, 46:53–61.

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

Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., and Xu, D. (2015). Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *ACM SIGSAC*, pages 146–157.

Scrivens, N. and Lin, X. (2017). Android digital forensics: data, extraction and analysis. In *ACM*, pages 1–10.

Shi, L., Fu, J., Guo, Z., and Ming, J. (2019). "jekyll and Hyde" is risky: Shared-everything threat mitigation in dual-instance apps. In *MOBISYS*, pages 222–235.

Srivastava, H. and Tapaswi, S. (2015). Logical acquisition and analysis of data from android mobile devices. *Information & Computer Security*.

Stefanko, Lukas (2020). 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/. Accessed: 24.03.2021.

Taubmann, B., Alabduljaleel, O., and Reiser, H. P. (2018). DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps. *Digital Investigation*, 26:S67–S76.

ThreatFabric (2020). 2020 - year of the RAT. https://www.threatfabric.com/blogs/2020_year_of_the_rat.html. Accessed: 24.03.2021.

Whittaker, Zack (2020). Eventbot: A new mobile banking trojan is born. https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born. Accessed: 24.03.2021.

Yang, S. J., Choi, J. H., Kim, K. B., and Chang, T. (2015). New acquisition method based on firmware update protocols for android smartphones. *Digital Investigation*, 14:S68–S76.