Real-Time Triggering of Android Memory Dumps for Stealthy Attack Investigation

Jennifer Bellizzi¹(✉) , Mark Vella¹ , Christian Colombo¹ ,
and Julio Hernandez-Castro²

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

Abstract. Attackers regularly target Android phones and come up with new ways to bypass detection mechanisms to achieve long-term stealth on a victim's phone. One way attackers do this is by leveraging critical benign app functionality to carry out specific attacks.

In this paper, we present a novel generalised framework, JIT-MF (*Just-in-time Memory Forensics*), which aims to address the problem of timely collection of short-lived evidence in volatile memory to solve the stealthiest of Android attacks. The main components of this framework are i) Identification of critical data objects in memory linked with critical *benign* application steps that may be misused by an attacker; and ii) Careful selection of trigger points, which identify when memory dumps should be taken during benign app execution.

The effectiveness and cost of trigger point selection, a cornerstone of this framework, are evaluated in a preliminary qualitative study using Telegram and Pushbullet as the victim apps targeted by stealthy malware. Our study identifies that JIT-MF is successful in dumping critical data objects on time, providing evidence that eludes all other forensic sources. Experimentation offers insight into identifying categories of trigger points that can strike a balance between the effort required for selection and the resulting effectiveness and storage costs. Several optimisation measures for the JIT-MF tools are presented, considering the typical resource constraints of Android devices.

Keywords: Memory forensics · Android security · Digital forensics · Incident response · Accessibility attacks

1 Introduction

Android has established itself as a leader in the mobile OS market [14], making it a primary target for malware. Whereas several detection mechanisms exist in the Google Play Protect suite [7], both to hinder the availability of malicious apps as well as to provide on-device detection, evasion techniques are still widely used, from obfuscation to stealthy execution.

© The Author(s) 2021

M. Asplund and S. Nadjm-Tehrani (Eds.): NordSec 2020, LNCS 12556, pp. 20–36, 2021.

https://doi.org/10.1007/978-3-030-70852-8_2

Accessibility services misuse in Android has emerged as a predominant stealth technique in recent years, primarily adopted by accessibility trojans pulling off phishing attacks in a particularly stealthy manner [2, 3, 6]. While initially proposed as a way to maliciously interact with victim apps in a stealthy way requiring only accessibility and overlay-related permissions [12], more recent work suggested that the level of stealth can be increased further by offloading most or all of the attack steps to benign apps [22]. In this setting, any classifier-based malware detector is fooled since critical attack steps are executed solely via white-listed victim apps. For instance, in the case of a messaging hijack attack, whereby an attacker aims to hide behind a victim’s identity to send a message, or intercept conversations from the victim’s phone, a malware may simply request accessibility permission and leverage other existing (or secretly installed) apps on the phone to read or send messages through that benign app.

Once the detection layer is breached, mitigation responsibility is shifted to incident response, where the use of digital forensics tools is central. In the case of such stealthy attacks, it becomes paramount to recreate the intrusion scenario by identifying the main attack steps. In the case of messaging hijacks, these comprise legitimate message sending or receiving/reading functionality, this time attacker-controlled. Evidence uncovering the critical attack steps is akin to an application logging its primary functionality. However, the absence of such fine-grained audit trails, which is usually the case, leaves investigators with no evidence in non-volatile storage to work with. Evidence collected from volatile memory becomes essential. While forensics tools that operate similarly have shown promise within very narrow domains, one cannot underestimate the significant challenge of dealing with short-lived evidence [15, 21, 22].

In this work, we aim to harmonise the approach taken by these individual tools into a generalised framework, focusing specifically on the challenge of timely memory dumps from benign victim apps through the careful selection of trigger points. While we present the *Just-in-Time Memory Forensics* (JIT-MF) framework within an accessibility misuse messaging hijack setting, the proposed concept extends to the general case of attacks carried out largely through benign apps. JIT-MF’s underpinning principles that distinguish it from state-of-the-art memory forensics tools are: i) Real-time collection of critical *data objects in volatile memory* related to the critical attack steps from 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 the collection of evidence from misused benign victim apps (rather than malware).

Evidence objects and trigger points are specific to investigation scenario-victim app pairs, as defined within *JIT-MF Drivers*. Four real-world case studies presented in this paper provide further insight into how to proceed from framework to tool implementation. This mainly revolves around the creation of

JIT-MF drivers. All cases concern messaging hijacks involving Pushbullet¹ and Telegram², covering SMS and instant messaging (IM).

Experimentation results from these case studies show that evidence object identification should focus on those data structures related to app functionality that are most likely to serve as critical attack steps. As for trigger points, we identified different candidate categories, ranging from those requiring general knowledge of the Android framework to ones requiring more in-depth knowledge of specific apps. Yet, results show that those requiring only Android framework knowledge are sufficiently effective. Furthermore, experimentation that focuses on the optimised implementation of JIT-MF tools shows that storage is a valid concern, especially for devices with limited resources and propose an approach to collect the specific objects through interactions with the Android’s runtime Garbage Collector. The key contributions of our work are:

- We introduce the concept of JIT-MF as a generic framework for memory forensic tools concerning Android attacks that offload their critical steps to benign apps.
- Provide insight into trigger point selection, a fundamental aspect to JIT-MF.
- Experimentation using four case studies that provide insight into developing practical JIT-MF tools.

2 Background and Motivation

2.1 Stealthy Android Accessibility Attacks

The misuse of accessibility services is on the increase in Android malware. Early instances [2] 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. In the case of Gustuff [2] this was done to perform banking transactions. More recently, however, with malware such as Eventbot [6] and BlackRock [5], this misuse has shifted from being leveraged to perform the actual attack to being used to maintain stealth. In the case of both Eventbot and BlackRock, the only permission requested upon installation is that of accessibility. The rest of the permissions required to perform the attack are obtained through the accessibility permission granted by the user. Malware developers can also exploit accessibility to leverage critical benign app functionality that coincides with the features they need. For instance, attackers who are motivated to create a malicious app to send SMSs via another phone to hide their identity (SMS crime-proxy), may exploit accessibility to silently install a benign legitimate SMSonPC app, e.g. Pushbullet [22], whose normal usage involves proxying sent/received SMSs through a remote PC. By signing up with phished credentials, as part of the setting up (step 1 of Fig. 1b) on the installed app, attackers gain full control over every SMS that is received and can send SMS remotely through a benign app, hiding its tracks and increasing the stealth level of the attackers’ subsequent steps.

¹ <https://www.pushbullet.com>.

² <https://telegram.org>.

Fig. 1. Misusing accessibility in different ways to carry out an SMS hijack attack.

Fig. 2. Instrumenting code via `ArtMethod` entry point.

2.2 Evidence Collection

Android Runtime (ART) has been the main managed runtime used by applications on Android [4] since it was released with Android KitKat in 2013 [23]. Similar to how JVM operates, ART uses two separate memory spaces to store application data; the stack and the heap [20]. Short-lived data objects of a running app, critical to attack steps, are found in volatile memory within the application's heap, managed by the Android Runtime.

Out of the box, ART provides functionality through which developers can dump heap data in the standard format of an `hprof` file, mainly for debugging purposes. The Java API equivalent for this is `Debug.dumpHprofData`³. A typical heap dump is semantically rich, containing information about an app's memory contents at the time the dump was taken. Most importantly, in our case, it includes information on the objects used and created by the app [4].

Another feature of ART is that of garbage collection. Figure 2 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 [4]. To do so, the GC uses a function exported by ART's binary module (`libart.so`), `Heap::GetInstances`, which has an object type filter, allowing the GC to filter on specific objects in memory. While convenient

³ <https://developer.android.com/reference/android/os/Debug#dumpHprofData>.

for selective evidence collection, the downside is that this function is not part of the public API and therefore may change unexpectedly between versions.

2.3 Android App Instrumentation

The Android OS uses APKs (Android Package Kit) as a package file format for distributing and installing mobile apps. The typical make up of an apk file consists of: an Android Manifest file providing essential information about the app, Dalvik (managed) bytecode in `classes.dex`, a `lib` directory for native code (e.g. ARM instructions), and other resources such as images/files required by the app. Native code can access the Android framework through the Java Native Interface (JNI), which enables the switching between native code and Dalvik bytecode. Therefore, since native code also calls into the Android framework, by using this framework to facilitate interception, we would also be able to intercept native code that calls into it.

ART uses specific C++ classes to mirror Java classes, their methods and associated instances, specifically using `Class`, `Object` and `ArtMethod` data structures respectively, as shown in Fig. 2 [10]. The `ArtMethod` data structure contains all the information about a particular Java method (method descriptors), such as the modifier, the class in which it is declared and the entry address of the method's code. Figure 2 shows how method hooking can be attained through `ArtMethod` patching, by first setting the method as native (Step 1), followed by entry point patching (Step 2), completing control-flow re-direction to instrumentation code.

3 JIT-MF

We assume the context of an investigation scenario whereby the device owner is not a perpetrator but a victim of a potential accessibility misuse attack targeting stealth. This may be the case with high-ranking government officials, or even high-profile business owners, as was the case in a report published earlier on this year [8]. In such cases, victims are expected to collaborate with forensic investigators to obtain critical evidence in the case of such an attack.

Our main aim is to obtain evidence, in the form of data objects, corresponding to critical application functionality from volatile memory which otherwise won't be made available by other sources of evidence. Due to their ephemerality, which is typical of app-level data objects, timely collection of such objects becomes critical. The primary goal behind the concept of Just-in-Time Memory Forensics (JIT-MF) is to extend the notion of memory forensics. We refer specifically to the kind that is carried out in real-time, over live process memory, capturing evidence associated with critical benign app misuse steps in a just-in-time fashion.

While the identification of evidence object(s) revolves around critical app functionality central to the threat in question, their timely collection requires the selection of *trigger points* which is a concept that is somewhat novel to our approach and requires more insight. These trigger points are events that occur

during the app’s runtime at which JIT-MF will invoke a partial memory dump. If the selected trigger points do not coincide with the invocation of misused app functionality, critical attack evidence may be lost.

Figure 3 gives an overview of the steps involved when implementing a tool based on the JIT-MF framework. Once a benign app is identified as having critical application steps which can be misused by an attacker, the app is extracted from the device (using `adb`). The app is instrumented, possibly using a combination of static and dynamic tools (depending on whether the device is rooted), to include code which uses the capabilities provided by ART to dump memory at the identified trigger points. Once repackaged, the app is re-installed and set up on the user’s phone. The memory dumps collected over time would then be gathered by a forensic analyst to reconstruct the attack steps.

Fig. 3. JIT-MF steps.

3.1 Heuristic for Evidence Object and Trigger Point Selection

A typical memory dump contains all the objects created and/or being used by an app (both specific to the app as well as those specific 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 message objects supporting the execution of messaging functionality and which may be hijacked during eventual attacks.

The selection of trigger points is specific to the following two aspects: i) The attack scenario for which we want evidence to be collected, and ii) How the app itself operates. Attempting to define a method for trigger point selection requires full knowledge of the specific app being analysed (and its version at the time). Given that the majority of the apps being analysed are expected to be third-party, assuming comprehensive knowledge of the app’s codebase is not practical. Instead, we propose a heuristic which we have used across four case studies. 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 (e.g. Wi-Fi, 4G, etc.); or else iii) Any **object transformation** of some sort (e.g. display on screen etc.).

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

Trigger Point Categories. Although the operations identified as candidate trigger points are all potentially valid, their degree of specificity to the app may differ. 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 (evidence object) 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 `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 precise 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 *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 require 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. At first glance, the impact of this trade-off is not obvious. Therefore we dedicate significant experimentation effort on comparing trigger point categories as part of the case studies presented in Sect. 4, to provide the necessary insight into trigger point selection for eventual JIT-MF based tools.

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 & 3 rd party library APIs	Black-box	Android API calls
App specific APIs	White-box	API calls specific to the app

3.2 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 required/critical objects in memory. In this setting, evidence objects are collected during the dumping process itself in an *online* fashion. Both approaches are compatible with non-rooted devices.

While JIT-MF defines those common steps 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. Their implementation starts off the aforementioned evidence/trigger point selection heuristic along with any argument value restrictions identified and is completed with the selection of an appropriate evidence collection method. Figure 4 illustrates the involvement of these drivers in the JIT-MF framework.

Fig. 4. JIT-MF drivers.

4 Experimentation

To evaluate the effectiveness and runtime overheads imposed on forensically enhanced devices, we conducted a series of experiments. These had the following objectives: i) Demonstrate that JIT-MF tools can collect evidence on stealthy accessibility attacks, effectively amplifying their forensic footprint; and ii) Perform a comparative analysis of the different trigger point categories, based on accurate memory dump triggers and their associated overheads.

4.1 Setup

Four messaging hijack case studies were set up for experimentation purposes, encompassing SMS and IM: 1) SMS Crime-proxy, 2) SMS Spying, 3) IM Crime-proxy and 4) IM Spying. A crime-proxy attack involves an attacker proxying messages through a victim’s phone via a benign app. This could help to foil attribution of compromising communication, possibly even resulting in incorrect attribution to the device owner. Spying through unlawful message interception comprises of attackers spying on device owners’ messages threatening their privacy,

and possibly even their safety. SMS hijack case studies make use of Pushbullet, an SMSonPC app that provides remote access to a device’s SMS functionality, and more. SMSonPC could be smuggled as part of an attack for stealth, or else could be the target of an attack in case a device owner is already making use of it. Telegram, on the other hand, is the app chosen for the IM setting due to its large userbase. In all case studies, we assume that accessibility malware has been installed and granted the accessibility permission by a non-suspecting device owner. We also carry out performance tests to analyse overhead storage and runtime costs incurred on legitimate user activity.

All four attacks were implemented as extensions to the Metasploit’s Meterpreter for Android⁴. For SMS-related attacks, the accessibility malware typically first sets up a Pushbullet installation and signs in using phished credentials. The remaining attack steps to send messages make direct use of Pushbullet’s web portal, automated using Selenium⁵ whereas any incoming messages can be obtained from browser logs. Furthermore, after sending an SMS, the attack can delete the Pushbullet app for additional stealth. SMS conversations for interception were simulated using `adb emu send <number> <message>`. No message deletion ensued in this case. IM-related attacks required the malware’s permanence, interacting with Telegram’s IM sending and viewing functionality in a continuous manner. The malware makes use of overlays in order not to attract the device owner’s attention. In Telegram’s crime-proxy attack case study, all sent messages are deleted after sending. In contrast, in the spying case study, a new phone with a different SIM card was used to assume the role of the sender and `adb` input events were used to automate message sending and receiving.

The full setup comprises: Pushbullet v17.7.19; Telegram v6.1.1 instrumented with the trigger points described in Sect. 3.1; both installed on an Android 10 emulator equipped with Frida-server v12.8.20 for instrumentation. Both online and offline evidence collection methods are encoded within the instrumentation code as described in Sect. 3.2, leveraging Frida’s `Java.choose()` and Android’s `API Debug.dumpHprofData()` respectively. To measure runtime overheads, we analyse storage and execution time overheads of both apps during legitimate message sending and reading/retrieving activities. 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 memory dumps and take note of whether or not they were found. All attacks were repeated 10 times, since it sufficed to reach convergence for all measurements taken.

Trigger Points. Eight trigger points (TP) were chosen, per attack scenario, two for each category defined in Table 1, attempting to leverage all available candidate trigger points in terms of disk input/output, network send/receive and miscellaneous object transformations. The chosen TPs are listed in Table 2, where TP1 is either file/disk or object transformation-related, whereas TP2 is network-related.

⁴ <https://github.com/rapid7/metasploit-framework/tree/master/documentation/modules/payload/android>.

⁵ <https://selenium-python.readthedocs.io/>.

Where possible, we put filters on black-box trigger points, for better specificity. For instance, the app directory (in the case of device events) is specific to the app and obtained dynamically at runtime using `getApplicationContext().getFilesDir().getParent()` provided by the Android API (typically being `/data/data/pushbullet|telegram`). Incoming/outgoing network statistics were obtained using Android’s `TrafficStats` package to monitor an increase in either, depending on the use case. 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. Native runtime calls were restricted to trigger on specific scenarios by checking whether the type of the file descriptor passed as an argument is a TCP socket or a file.

Table 2. Trigger points selected.

Case study	TP Category	TP #	Event selected
(SMS) Pushbullet - Crime-proxy	Native	TP 1	<code>write()</code> - to disk
	Runtime	TP 2	<code>read()</code> - from socket
	Device Event	TP 1	Increase in app directory size
		TP 2	Increase in incoming network traffic
	Android & 3 rd Party APIs	TP 1	<code>android.content.ContentResolver.insert</code>
		TP 2	<code>android.telephony.SmsManager.sendTextMessage</code>
	App specific APIs	TP 1	<code>com.pushbullet.android.sms.SmsSyncService.a</code>
		TP 2	<code>com.pushbullet.android.providers.syncables.SyncablesProvider.insert</code>
(SMS) Pushbullet - Spying	Native	TP 1	<code>read()</code> - from disk
	Runtime	TP 2	<code>write()</code> - to TCP socket
	Device Event	TP 1	Increase in app directory size
		TP 2	Increase in outgoing network traffic
	Android & 3 rd Party APIs	TP 1	<code>android.content.ContentResolver.registerContentObserver</code>
		TP 2	<code>com.google.android.gms.gcm.GcmReceiver.onReceive</code>
	App specific APIs	TP 1	<code>com.pushbullet.android.sms.SmsSyncService.a</code>
		TP 2	<code>com.pushbullet.android.gcm.GcmService.a</code>
(IM) Telegram - Crime-proxy	Native	TP 1	<code>File.open()</code>
	Runtime	TP 2	<code>send()</code> - to socket
	Device Event	TP 1	Increase in app directory size
		TP 2	Increase in outgoing network traffic
	Android & 3 rd Party APIs	TP 1	<code>android.widget.EditText.setText</code>
		TP 2	<code>android.app.SharedPreferencesImpl\$EditorImpl.commitToMemory</code>
	App specific APIs	TP 1	<code>org.telegram.tgnet.ConnectionsManager.native_sendRequest</code>
		TP 2	<code>org.telegram.messenger.SendMessagesHelper.performSendMessageRequest</code>
(IM) Telegram - Spying	Native	TP 1	<code>File.open()</code>
	Runtime	TP 2	<code>recv()</code> - from socket
	Device Event	TP 1	Increase in app directory size
		TP 2	Increase in incoming network traffic
	Android & 3 rd Party APIs	TP 1	<code>android.view.ViewGroup.dispatchGetDisplayList</code>
		TP 2	<code>android.app.ContextImpl.sendBroadcast</code>
	App specific APIs	TP 1	<code>org.telegram.ui.Cells.DialogCell.update</code>
		TP 2	<code>org.telegram.messenger.MessagesStorage.putMessages</code>

4.2 Results

Effectiveness. Table 3 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 presented show the effectiveness obtained by using both offline and online collection methods which, as can be observed from the table, have very similar results. Looking at the results column-wise, i.e. across trigger point categories, results show that the hypothesis that white-box trigger points are more accurate than their black-box counterparts does not hold. Having said that, when results are analysed row-wise, i.e. across attacks, in each case study there is at least one trigger point that returns a 100% accuracy. For both Pushbullet and Telegram 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.

Overall, results from this small, albeit representative, number of case studies show that while identification of entirely accurate trigger points is possible, this is not at all straightforward. This, of course, merits further investigation since the timely dumping of evidence is central to JIT-MF. On the upside, it looks like selecting accurate trigger points could be possible solely within the black-box categories, which are those requiring minimal app-specific knowledge.

To further make a case for the JIT-MF framework, we compare the evidence obtained by the JIT-MF tool (highlighted in Table 3), with that returned by typical (baseline) logs which feature in classical forensic analysis. For every attack scenario in the experiment, we obtain a copy of logcat at the point in time when an attack has occurred, we analyse network traffic and get sqlite database files which are used for on-device storage for both apps.

In the case of logcat we did not observe any of the metadata acquired by JIT-MF in any of the logs for all four attack scenarios. For Telegram, it was possible to instrument the app to enable verbose logging in logcat dynamically. However, this did not make any difference with regards to critical metadata present in logcat. As for sqlite files, in the case of Pushbullet, we could only observe the received SMS messages, but no history of their access. For sent messages, one would have to root the phone to obtain Android’s default message store `mmsms.db`. In the case of Telegram, being a cloud app, sqlite files only provide portions of cached data of received and sent messages. In the case that the attacker deletes the chat, no evidence of the sent messages is found at all. Furthermore, in the case of intercepted messages in Telegram, whereas there is a `state` field that indicates whether or not a particular message was read, it does not indicate the *time* at which the message was read. All network traffic related to Pushbullet and Telegram communication protocols was exchanged over HTTPS. Therefore while an initialised connection can be observed, none of the traffic is decipherable unless decryption keys are obtained.

Runtime overheads. Runtime overheads were obtained during normal usage of the app, by legitimately invoking events that would be misused by an attacker in the case of a messaging hijack attack. Figure 5 shows the storage requirements per trigger point category over the ten runs. Here we only consider the online collection method for the time being. Overall, storage requirements are tied to the number of times trigger points are hit per run. What is interesting to note is that the black-box categories can still be as frugal as their white-box

counterparts, showing that the use of filters paid off. As for the hprof-based offline method, we note that while the average dump size required by online collection is around 143 kB, that required by the offline method is 203 MB (an order of magnitude *more* on average), per attack scenarios and trigger point chosen. Execution overheads associated with memory-dumping instrumentation code were negligible for both collection methods in Telegram’s case, with an increase of 0.2 s at worst. For Pushbullet this value increases to 6 s at worst (20% of the cases) however given that Pushbullet operates from a browser setting, this execution overhead does not incur any lag on the phone’s main UI thread, enabling the user to continue using the phone normally.

4.3 Discussion

JIT-MF Amplifies the Forensic Footprint of Stealthy Attacks. Effectiveness results show that, while trigger point-dependent, key evidence related to stealthy messaging hijacks was only accessible through the JIT-MF approach. This is the central tenet of the approach. While at the code and network levels key evidence can be hidden through obfuscation and encryption, evidence linked to the key attack steps must be revealed in volatile memory, even if only for a brief time.

Table 3. Trigger point effectiveness: % accuracy over 10 runs.

Trigger point Category/Scenario		Crime-proxy - IM		Spying - IM		Crime-proxy - SMS		Spying - SMS	
		Online	Offline	Online	Offline	Online	Offline	Online	Offline
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 & 3 rd Party APIs	TP 1	90	90	100	100	100	100	0	0
	TP 2	80	60	100	100	100	100	80	80
App specific APIs	TP 1	100	100	100	100	0	0	0	0
	TP 2	0	0	60	60	100	100	80	80

Fig. 5. Storage overhead in bytes, per trigger point category, attack scenario and app.

Black-Box Trigger Point Categories Show Promise. While results show that selecting the right, most accurate, trigger point can be an arduous task, the fact that black-box ones can be as effective and efficient as white-box ones is good news. Obviously, this observation requires substantial follow-up; however, this bodes well for efforts attempting to automate tasks related to JIT-MF driver implementation, of which trigger point selection is central. While both effectiveness and runtime overheads so far do not favour any of the three black-box categories, it seems that certain trigger point categories might be less resource-intensive for some apps, and more for others. This, however, merits more investigation.

Optimising on 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 0.1 kB to be effective, these results must also be analysed in the context of practical JIT-MF tool deployment. 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. IM functionality could result in daily triggers on the order of hundreds to thousands. While 128GB smartphones are now the norm, users would rather use the space for smartphone functionality rather than to store forensic evidence.

In this respect, the suggested way-forward concerns enhancing the collection method as defined in JIT-MF drivers in two ways. Firstly, we propose to improve the collection method with a data transfer method. A transfer method should establish both the transport channel, e.g. SD card, `adb`, network etc., as well as the frequency of synchronisation points whenever applicable. Secondly, a sampling option should also be provided. Rather than collecting the entire evidence, successful incident response is possible even if only a subset of the attack steps are recorded. In the case of crime-proxy attack, for example, a fragment of a conversation could already provide sufficient clues pointing towards ongoing hijacks, full content disclosure would require further effort. Sampling may be carried out either periodically, e.g. sample maximum event objects per time-frame, or else on a rule basis, e.g. sample outgoing messaging objects based on their destination number, say those not found in the contact list. Ultimately, the right combination for JIT-MF collection depends on the sensitivity of the investigation context. For instance, in the context of a high-profile government agent, or Fortune 500 CEO, it could be worth spending extra money on high-spec devices to opt for a more resource-hungry collection method.

5 Limitations and Future Work

The primary contribution made by this early attempt to investigate stealthy Android attacks is a general framework, JIT-MF, upon which specific tools can be modelled. The four case studies presented here provide valuable insight concerning how to go about evidence object identification and trigger point selection. Therefore these results have to be understood within this limited scope. Moving

on to a larger-scale empirical study is undoubtedly going to require some level of automation. The experience derived from the manual process undertaken so far, as guided by the heuristic described in Sect. 3.1, presents a solid foundation for this crucial next step.

A natural progression of this research also concerns developing complete JIT-MF tools, initially those targeting messaging hijack investigation scenarios. In this context, JIT-MF tool development can begin with those evidence object and trigger point combinations that were already shown to be sufficiently effective. The pending work concerns: i) Target app repackaging, compatible with non-rooted devices; ii) JIT-MF driver enhancement with an extension for the collection method as discussed; iii) Correlation with additional evidence, e.g. Call Data Record logs and cloud back-ups to provide additional context for the collected evidence objects, within a forensic timeline. Comprehensive timelines can help give investigators a complete picture of events, and thus assist in discerning between hijack activity and normal device usage with device owner consent. Additionally, it would be interesting to assess the difference JIT-MF evidence makes on forensic timeline richness as compared to those produced using only state-of-the-art evidence collection.

Ultimately JIT-MF is not intended as a comprehensive solution. JIT-MF tools also need to pull robust implementation, as well as addressing instrumentation issues related to apps that perform code integrity checks. Despite the assumption of the device owner’s collaboration, privacy issues still abound and have to be taken care of. Finally, app instrumentation for memory dumping is incompatible with system apps on non-rooted phones.

6 Related Work

While our work focuses on the problem of accessibility misuse to aid stealthy attacks and builds on previous work in this regard [22], stealthy attacks aiming for persistence go beyond accessibility misuse. Other similar attack vectors include dual-instance apps [19] and stealthy persistent trojans like Triada [1] which evade common detection mechanisms.

Similar to monitors like REAPER [11] and MOSES [25], 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.

Saltaformaggio et al. [15–17] and Taubmann et al. [21] also developed tools which are after ephemeral data in memory, to reconstruct flows within an app’s runtime which can be critical in a forensic investigation. They do so by reconstructing critical data structures from memory dumps. Rather than within a general concept, their ephemeral data is very specific (GUI elements for screen flows and TLS private keys respectively). DroidKex [21] acquires memory dumps upon *send* and *receive* functionality of an app, an indicator that TLS connections are being established, similar to JIT-MF’s concept of trigger points.

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 for a broad range of investigation scenarios. The concept can be even applied to reconstructing timelines from specific log files using custom analysers [13].

Several works have tackled recovery and digital forensics of specific messaging apps, like Telegram [9, 18, 24]. The techniques mainly utilise disk images to retrieve valuable evidence. While stored data can be useful, it is up to app developers which metadata to store. Metadata critical to an investigative scenario may not be available at all from disk (as seen in the results). Even if it was, that may not be the case across versions. Furthermore, with the increase in popularity of cloud-based messaging apps, fewer data becomes available locally to retrieve.

7 Conclusions

Due to its ubiquitous presence, Android has become a significant target for malware. Recent studies show the existence and gradual increase of stealthy Android attacks that through accessibility, leverage benign app functionality to execute critical attack steps. Since such attacks offload the majority of their actions to benign apps, current techniques aimed at detecting malware based on the presence of malicious behaviour are rendered ineffective. Volatile memory remains the only place where evidence of such attacks may be found.

To address this problem, we introduce a framework called JIT-MF which, through carefully selected trigger points, forensically enhances apps to timely dump sections of memory that could contain critical data objects, as evidence. We evaluate this framework in the context of accessibility messaging hijack attacks, using widely deployed apps as victim apps. Results from four case studies show that: i) JIT-MF tools enhance the forensic footprint of stealthy attacks beyond the current baseline; ii) There is a category of trigger points that is both effective and only requires basic knowledge of the app; and iii) JIT-MF can be optimised for storage. In this paper, we shed light on the capabilities of JIT-MF in the context of messaging hijack attacks within Android. However, the framework can be extended to cater for other investigative scenarios and even operating systems, to capture evidence that would otherwise be irreparably lost.

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

References

1. Triada: organized crime on Android (2016). <https://www.kaspersky.com/blog/triada-trojan/11481/>
2. Gustuff: Weapon of Mass Infection (2019). <https://www.group-ib.com/blog/gustuff>
3. Year of the RAT (2020). https://www.threatfabric.com/blogs/2020_year_of_the_rat.html

4. Android Developer's Guide (2020). <https://developer.android.com/>
5. BlackRock - The Trojan that wanted to get them all (2020). <https://www.threatfabric.com/blogs/blackrock-the-trojan-that-wanted-to-get-them-all.html>
6. Eventbot: A new mobile banking trojan is born (2020). <https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born>
7. Google Play Protect (2020). <https://www.android.com/play-protect/>
8. Jeff Bezos hack: Saudi Arabia calls claim 'absurd' (2020). <https://www.bbc.com/news/business-51171400>
9. Azhar, M.A.H.B., Barton, T.E.A.: Forensic analysis of secure ephemeral messaging applications on android platforms. In: Jahankhani, H., et al. (eds.) ICGS3 2017. CCIS, vol. 630, pp. 27–41. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-51064-4_3
10. Costamagna, V., Zheng, C.: ARTDroid: a virtual-method hooking framework on android art runtime. In: ESSoS, pp. 20–28. IMPS (2016)
11. Diamantaris, M., Papadopoulos, E., Markatos, E., Ioannidis, S., Polakis, J.: Reaper: real-time app analysis for augmenting the android permission system. In: CODASPY, pp. 37–48. ACM (2019)
12. Fratantonio, Y., Qian, C., Chung, S.P., Lee, W.: Cloak and dagger: from two permissions to complete control of the UI feedback loop. In: IEEE S&P, pp. 1041–1057. IEEE (2017)
13. Hargreaves, C., Patterson, J.: An automated timeline reconstruction approach for digital forensic investigations. *Digital Invest.* **9**, S69–S79 (2012)
14. L., S.: Android - Statistics & Facts (2020). <https://www.statista.com/topics/876/android/>
15. Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D.: GUITAR: Piecing together android app guis from memory images. In: CCS, pp. 120–132. ACM SIGSAC (2015)
16. Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D.: VCR: app-agnostic recovery of photographic evidence from android device memory images. In: CCS, pp. 146–157. ACM SIGSAC (2015)
17. Saltaformaggio, B., Bhatia, R., Zhang, X., Xu, D., Richard, G.: Screen after previous screens: spatial-temporal recreation of android app displays from memory images. In: USENIX, pp. 1137–1151 (2016)
18. Satrya, G., Daely, P., Nugroho, M.: Digital forensic analysis of telegram messenger on android devices. In: ICTS, pp. 1–7. IEEE (2016)
19. Shi, L., Fu, J., Guo, Z., Ming, J.: “Jekyll and Hyde” is Risky: shared-everything threat mitigation in dual-instance apps. In: MobiSys, pp. 222–235. ACM (2019)
20. Surin, T.: Dealing with Large Memory Requirements on Android (2020). <https://pspdfkit.com/blog/2019/android-large-memory-requirements/>
21. Taubmann, B., Alabduljaleel, O., Reiser, H.: DroidKex: fast extraction of ephemeral TLS keys from the memory of android apps. *Digital Invest.* **26**, S67–S76 (2018)
22. Vella, M., Rudramurthy, V.: Volatile memory-centric investigation of SMS-hijacked phones: a Pushbullet case study. In: FedCSIS, pp. 607–616. IEEE (2018)
23. Vitas, M.: ART vs Dalvik Introducing the New Android Runtime in KitKat (2020). <https://infinum.com/the-capsized-eight/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat>
24. Zhang, L., Yu, F., Ji, Q.: The forensic analysis of WeChat message. In: IMCCC, pp. 500–503. IEEE (2016)
25. Zhauniarovich, Y., Russello, G., Conti, M., Crispo, B., Fernandes, E.: Moses: supporting and enforcing security profiles on smartphones. *IEEE TDSC* **11**(3), 211–223 (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.