Journal of Cybersecurity and Privacy

Article

VEDRANDO: A Novel Way to Reveal Stealthy Attack Steps on Android through Memory Forensics

Jennifer Bellizzi [1,*], Eleonora Losiouk [2], Mauro Conti [2], Christian Colombo [1] and Mark Vella [1]

[1] Department of Computer Science, University of Malta, MSD 2080 Msida, Malta; christian.colombo@um.edu.mt (C.C.); mark.vella@um.edu.mt (M.V.)
[2] Department of Mathematics, University of Padua, Via Trieste 63, 35121 Padua, Italy; elosiouk@math.unipd.it (E.L.); mauro.conti@unipd.it (M.C.)
* Correspondence: jennifer.bellizzi@um.edu.mt

Abstract: The ubiquity of Android smartphones makes them targets of sophisticated malware, which maintain long-term stealth, particularly by offloading attack steps to benign apps. Such malware leaves little to no trace in logs, and the attack steps become difficult to discern from benign app functionality. Endpoint detection and response (EDR) systems provide live forensic capabilities that enable anomaly detection techniques to detect anomalous behavior in application logs after an app hijack. However, this presents a challenge, as state-of-the-art EDRs rely on device and third-party application logs, which may not include evidence of attack steps, thus prohibiting anomaly detection techniques from exposing anomalous behavior. While, theoretically, all the evidence resides in volatile memory, its ephemerality necessitates timely collection, and its extraction requires device rooting or app repackaging. We present VEDRANDO, an enhanced EDR for Android that accomplishes (i) the challenge of timely collection of volatile memory artefacts and (ii) the detection of a class of stealthy attacks that hijack benign applications. VEDRANDO leverages memory forensics and app virtualization techniques to collect timely evidence from memory, which allows uncovering attack steps currently uncollected by the state-of-the-art EDRs. The results showed that, with less than 5% CPU overhead compared to normal usage, VEDRANDO could uniquely collect and fully reconstruct the stealthy attack steps of ten realistic messaging hijack attacks using standard anomaly detection techniques, without requiring device or app modification.

Keywords: remote live forensics; mobile security; app virtualization

check for updates

Citation: Bellizzi, J.; Losiouk, E.; Conti, M.; Colombo, C.; Vella, M. VEDRANDO: A Novel Way to Reveal Stealthy Attack Steps on Android through Memory Forensics. *J. Cybersecur. Priv.* **2023**, 3, 364–395. https://doi.org/10.3390/jcp3030019

Academic Editors: Mario Antunes and Carlos Rabadão

Received: 4 May 2023
Revised: 25 June 2023
Accepted: 5 July 2023
Published: 10 July 2023

1. Introduction

Malicious app developers constantly seek ways to evade security measures and detection techniques through stealthy attack vectors that lure mobile users into downloading malicious apps onto their devices and that prolong the malware's lifetime once on the device. One way Android malware achieves stealth is by disguising its activities as legitimate. This not only enables the spread of malware on the official Google Playstore [1–3] but also *allows the malware to evade detection on the victim's device*, resulting in devastating effects (such as the unauthorized transfer of funds from legitimate banking apps), whenever attacks are noticed only from the consequences of their successful execution [4–9]. Established attack vectors such as accessibility [10] and several others [11–16] allow malware to attain stealth by leveraging living-off-the-land tactics that enable malware to offload critical attack steps to benign legitimate app functionality. For instance, the benign functionality of sensitive app categories such as messaging and financial apps can be instrumental to stealthy attacks aiming to hijack this functionality to offload attack steps such as malware propagation through messaging or seemingly legitimate fund transfers.

Due to their similarity with benign functionality, attacks that hijack this benign functionality render threat detection mechanisms useless. Furthermore, this level of stealth typically leads to victims raising the alarm and initiating an investigation process when the consequences of the attack are evident (e.g., missing funds), which occurs way after the attack steps have been carried out (late detection). Incident responders and security operations center (SOC) analysts investigating such incidents must derive the covert nature of these stealthy attacks from their deliberately small footprint [17,18]. Regardless of the stealthiness of an attack, however, its execution must occur in memory [19,20]. Therefore, in stealthy attack scenarios, memory forensics becomes crucial to recover key artefacts in memory that may disclose stealthy attack steps and provide a better context for investigators to reconstruct the attack steps. Specifically, for the class of stealthy attacks that hijack benign app functionality, resulting in late detection due to their stealth, previous research showed that the associated in-memory evidence is ephemeral and requires *timely* memory collection [21].

The standard enterprise threat solution is endpoint detection and response (EDR). EDR tools monitor and record events occurring on endpoints (devices, PCs, servers, etc.), providing security teams with the necessary visibility to investigate and mitigate threats through advanced threat detection, investigation, and response capabilities [22]. EDRs typically leverage known malicious tactics, techniques, and procedures (TTPs) or behavioral analytics to detect unusual attack-related behavior. However, this form of threat detection falls short when dealing with novel stealthy attacks whose tactics are unknown or cannot be distinguished from legitimate benign app interactions. Even if EDRs cannot detect stealthy attacks that leverage benign app functionality, they can provide a fallback through live forensics. EDRs can collect evidence from the device and applications using functionality the underlying OS exposes through Android APIs. However, EDRs must rely on third-party application logs, which may not contain the necessary evidence to disclose and reconstruct attack steps. While memory forensics could compensate for this limitation, this presents a challenge, due to the restrictions on unrooted devices and the non-extendibility of stock Android kernels in mobile devices.

Just-in-time memory forensics (JIT-MF) [21] is an experimental technique that uses app repackaging as an alternative for dumping evidence from memory using stock Android devices. While avoiding the need for device rooting, JIT-MF still requires significant reverse engineering effort for app repackaging, which is time-consuming and renders the technique invasive and infeasible when considering the large number of sensitive apps that could be hijacked. Its feasibility regarding the customization needed for each hijack-targeted app poses another challenge for adoption. Furthermore, while previous work [21,23,24] demonstrated how JIT-MF could uniquely collect the activity of hijacked apps directly from volatile memory, its value in an investigation setting for attack step detection has not been shown.

In this paper, we present *VEDRANDO* (i.e., Volatile-memory-enhanced EDR for ANDrOid) an enhanced EDR for Android that allows the timely collection of challenging volatile memory artefacts and the detection of stealthy attacks that hijack benign applications. VEDRANDO has two main components: an events collector, and an attack detector. The events collector component collects elusive evidence of stealthy attacks that is not found in other forensic sources, by employing a state-of-the-art Android EDR tool with experimental memory forensics (JIT-MF), thus improving the state of the art for EDR Android tools by allowing the timely collection of forensic sources from memory. This component addresses existing feasibility and implementation challenges by leveraging JIT-MF infrastructure-based drivers and app virtualization techniques to ease the burden of app-specific JIT-MF driver development, while avoiding device rooting and app repackaging. The attack detector component uses a detection algorithm that, given the additional evidence collected from the events collector component, can detect and expose the hidden, attack-related behavior of benign app hijack attacks using standard anomaly detection methods, resulting in complete and accurate attack step reconstruction.

Our evaluation extends previous work [25], showing that JIT-MF infrastructure-based drivers ensure the feasibility of our approach, as these drivers are reusable over 92.2% of the 550 most popular Android apps (ranked by all-time number of downloads on App-

Brain [26] according to GooglePlay statistics) when leveraging SQLite libraries. We assessed the performance overheads of VEDRANDO by conducting a runtime evaluation of the solution using theUI Exerciser Monkey tool to simulate normal traffic on a set of apps from the most popular 100 apps in the Google PlayStore (as listed on AppBrain), which were not previously installed on the phone or manufacturer-specific (33 apps in total). Our results showed that VEDRANDO worked with 84.8% of the apps, with negligible overheads (less than a 5% CPU usage increase), while being feasible and minimally invasive by avoiding device rooting, app repackaging, and additional app reverse-engineering overheads. Finally, we demonstrated the value of our solution through a series of investigation case study setups, involving stealth messaging hijack attacks targeting ten popular instant messaging (IM) apps. The results from the ten case studies showed that VEDRANDO can uniquely collect critical evidence from memory. Furthermore, VEDRANDO's attack detector could fully reconstruct stealthy attack steps, including the malware entry point in all case studies, for a combination of anomaly detection methods and inputs. In summary, we make the following contributions:

- We introduce VEDRANDO, a novel Android EDR that addresses the challenge of the timely collection of volatile memory artefacts and the detection of stealth attacks that hijack benign applications. VEDRANDO leverages JIT-MF for memory forensics capabilities. It addresses existing feasibility challenges regarding JIT-MF drivers' app-specific customization and installation through generic infrastructure-based JIT-MF drivers and app-level virtualization, resulting in a solution that avoids app reverse-engineering, device rooting and app repackaging;
- We conducted a runtime evaluation of VEDRANDO's events collector across 33 apps, achieving an 84.8% success rate in running apps within VEDRANDO's setup and introducing an average increment up to 4.9 percentage points (pp) in CPU usage and 0.7 pp in consumed memory compared to the app's typical performance;
- We demonstrated the value of VEDRANDO in the context of ten messaging hijack attack investigations of popular Android IM apps. Our results showed that VEDRANDO could disclose evidence of attack steps not collected by state-of-the-art EDRs for all case studies. Once collected, VEDRANDO could detect these events as anomalous using existing anomaly detection methods and reconstruct all attack steps.

2. Background

2.1. Endpoint Detection and Response Systems (EDR)

EDR tools monitor activities on endpoints (devices, PCs, servers, etc.) and provide alerts when potentially malicious behaviors are observed. These tools typically comprise a control server and EDR clients deployed on endpoints, which communicate with the server to send logs of ongoing events and receive instructions for threat monitoring purposes. Potential threats on endpoint systems are flagged if events gathered from EDR clients match manually crafted expert rules from a knowledge base describing a low-level attack pattern. In the case of incidents, EDRs provide investigators with live forensics capabilities, by collecting evidence from endpoints, which investigators use for threat hunting and reconstructing the malware's attack steps, to enable a response.

State-of-the-art EDRs rely on services provided by the underlying operating system (OS) to collect forensic sources. In the case of mobile phones, forensic sources for the investigation of an attack comprise application data, data related to services (e.g., connectivity, telephony), system data (e.g., packages installed, user data), and data in external storage [27]. Table 1 shows a complete list of sources that can be collected from Android devices by Android EDR tools. While present on the device, collecting some forensic artefacts requires system privileges (rooted device), which is impossible in the enterprise setting where stock Android devices and apps are used. Furthermore, dumping of evidence from memory due to insufficient evidence from application logs is not possible with the state-of-the-art Android EDR tools, as this requires functionality that the underlying Android OS does not provide.

Table 1. Forensic artefact source collection comparison of Android Forensic EDR tools, as shown in [27], as of 2021. Commercial tools in this study included: Oxygen Forensic Suite, MOBILedit, XRY Forensic Examiner's Kit, UFED, and EnCase Mobile Investigator

	Artefact Source	**SystemSens** [28]	**DroidWatch** [29]	**Device Analyzer** [30]	**AFLogical OSE** [31]	**DELTA** [32]	**Andriller** [33]	**Commercial Tools**	**ReLF** [27]
System	OS info			✓			*	*	✓
	Hardware info			✓			*	*	✓
	System settings			✓			✓	*	✓
	Battery statistic	✓		✓		✓		*	✓
App	Installed packages		✓	✓		✓	✓	*	✓
	Running processes	✓		✓		✓		*	*
Telephony	Contacts		✓	✓	✓		*	*	✓
	Call logs	✓	✓	✓	✓	✓	*	*	✓
	SMS/MMS	✓	✓	✓	✓	✓	*	*	✓
	SIMs & subscriptions			✓			*	*	✓
	Cellular info	✓		✓		✓		*	✓
Connectivity	Wi-Fi status & hotspots	✓		✓		✓	✓	*	✓
	Bluetooth info			✓		✓		*	✓
	NFC status							*	✓
	VPN profiles								*
	NIC info & Netstat	✓		✓				*	*
Storage	Storage volume info			✓		✓		*	✓
	Filesystem & file stats			✓		✓		*	*
	Retrieve arbitrary file			✓		✓		*	*
Sensors	Location		✓	✓		✓		*	✓
	Microphone					✓		*	
	Sensor info & logging			✓		✓		*	✓
User Data	User accounts		✓				*	*	✓
	Device user profiles					*		*	*
	Calender	✓	✓				*	*	
	Browser history		✓				*	*	*
Other	Screen state & capture	✓	✓	✓		*	*	*	
	Key & touch logging					*			
	Remote logging	✓	✓	✓		✓			✓

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

2.1.1. App-Specific Forensic Sources

In the case of stealth attacks that hijack benign app functionality, a critical forensic source that could contribute towards detecting attack-related behavior are the logs of hijacked benign app events. These are typically found in app databases in the app data directory of Android internal storage (`/data/data/<package_name>`), which is inaccessible to users and forensic investigators unless the device is rooted.

While app developers can make these logs available to users through different APIs or backup functionality, the output may still be insufficient to disclose app hijack attacks. The content of these logs is at the discretion of the app developers, whose interests may align more with the app's usability rather than logging events that could indicate an app hijack. Therefore, even after collecting this evidence, investigators cannot detect attack-related behavior to reconstruct the complete stealth attack steps.

2.1.2. Event Reconstruction

When the logs and evidence from different forensic sources have been collected, investigators are left with a list of various events that occurred in the system, which may lead to a cognitive overload. Therefore, several works aimed to solve this problem by finding ways to combine multiple "low-level events" comprising individual log entries in forensic sources into fewer "high-level events" that are human-understandable events, which could be more instructive for investigators to reconstruct the attack steps. Generalized automated solutions through event pattern matching [34], semantic-based correlation [35], and ontology-based techniques [36] exist. However, these are dependent on the available forensic sources. Therefore, reconstructing events from individual low-level log events remains an open research problem [37,38] that must be addressed depending on the investigator's available forensic sources.

2.2. Just-in-Time Memory Forensics (JIT-MF)

JIT-MF [21,23,24] is an experimental technique conceived to be adopted by incident response tools for stock smartphones without breaking any security controls. Rather, given that its primary purpose is to aid device owners in recovering stealth attack steps, it assumes the device owner's collaboration.

It tackles the problem of missing evidence in application logs by enabling the enhancement of Android apps with app-specific logging through JIT-MF Drivers that dump evidence from memory, using app instrumentation, to avoid device rooting. These drivers are installed within third-party applications to log additional data from the memory to storage as part of the forensic readiness stage of incident response, without changing the application code beyond hooking. While JIT-MF drivers drive the process of an active collection of app artefacts from memory, as shown in Figure 1, the JIT-MF driver runtime provides any services required by the JIT-MF driver to operate; that is, the ability to register triggers, allow access and retrieval of artefacts from process memory, and move generated output to storage.

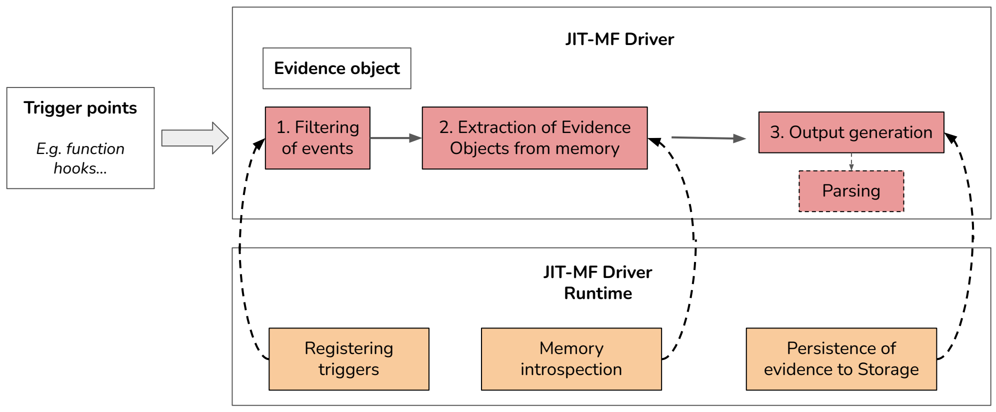


Figure 1. JIT-MF Concept.

JIT-MF Drivers have two main properties: *Evidence_objects* are identified as those application-specific objects whose presence in memory implies the execution of some specific app functionality, possibly a delegated attack step. *Trigger_points* define which application instructions indicate that *Evidence_objects* are in memory, and hence when memory dumps should be triggered. Therefore, trigger points are crucial for the timely dumping of evidence objects in memory that could aid in reconstructing stealthy attack steps. Previous works [21,23,24] extensively explored the reliability of JIT-MF from different aspects, including: (i) comparing *trigger_point* placement in different layers of the Android technology stack; (ii) using different sampling strategies for app stability; and (iii) identifying different methods to for developing a JIT-MF Driver.

JIT-MF Logs

Evidence_objects in memory retrieved on the invocation of *Trigger_points* are stored in `*.jitmflog` files. Taking the case of a stealth messaging hijack attack as an example, attackers aim to send messages through a victim's benign messaging app and immediately delete them, to ensure that the victim remains unaware of the malicious attack step. Benign messaging apps can be enhanced with a JIT-MF driver whose *evidence_object* is defined as the application-specific *MessageObject* containing details of the message sent, and the *trigger_point* can be defined as any operation handling the *MessageObject*. A sample of the resulting `*.jitmflog` file, is shown in Listing 1, whereby each entry consists of a timestamp of the entry, the event type, and the metadata derived from the evidence object (in this case *MessageObject*) as dumped from the memory at a particular trigger point.

The application-specific *MessageObject* contains the properties that populate a log entry with the necessary information regarding an event. However, identifying this object in the first place is challenging, since (i) application developers may change the properties of this object or even the object name itself when newer versions of the application are released, and (ii) different messaging applications use different *MessageObjects* with different properties.

The app-specific nature of JIT-MF drivers makes their development tedious. They require prior app-specific knowledge, typically gained through code comprehension in the case of open-source apps and app reverse engineering when dealing with closed-source apps. Both options are infeasible considering the growing number of apps in Android app play stores. Furthermore, while foregoing the requirement of device rooting, app repackaging does not comply with enterprise standards for use of stock Android apps.

Listing 1. JIT-MF log entry sample generated while using WhatsApp, Telegram, and Signal Android apps [21].

```
{"time": "1662482712", "event": "Whatsapp Message Sent", "trigger_point": "android.database.sqlite.SQLiteDatabase",
      "object": {"date": "", "message_id": "4138821D2BF18D844720CBBF5067A5AD,", "text": "Normal_message_1", "to_id":
      "7196@s.whatsapp.net]", "to_name": "", "to_phone": "", "from_id": "", "from_name": "", "from_phone": ""}}
{"time": "1662485256", "event": "Telegram Message Present", "trigger_point": "recv", "object": {"date": "1662483779",
       "message_id": "2328", "text": "Normal_message_1", "to_id": "5181266731", "to_name": "target_phone;;;",
      "to_phone": "35699626972", "from_id": "1679923803", "from_name": "contact_phone;;;", "from_phone":
      "35679247196"}}
{"time": "1662487182", "event": "Signal Message Present", "trigger_point": "open", "object": {"date": "1662487132503"
      , "message_id": "168", "text": "Normal_message_1", "to_id": "RecipientId::2", "to_name": "null", "to_phone": "
      +35699626972", "from_id": "RecipientId::3", "from_name": "null", "from_phone": "+35679247196"}}
```

2.3. *App-Level Virtualization*

To date, JIT-MF experimentation has relied on app repackaging to avoid device rooting, which enables the technique to be used on stock Android devices. Android app-level virtualization has emerged as a new technique that can load arbitrary third-party APKs without installation or modification, providing a possible solution to app repackaging. This enables an app (container) to create a virtual environment where other app (plugins) can run. Plugins can execute independently from the underlying Android OS and other virtual environments. DroidPlugin [39] and VirtualApp [40] are the two most well-known frameworks supporting the generation of Android virtual environments and share a common design, as shown in Figure 2. A virtual environment can run any Android app, single or multiple apps at a time, and apps not installed on the device, without requiring any additional privileges being enabled on the device (e.g., root privileges).

Figure 2 shows how the container app loads and runs plugin apps through a proxy. The container app intercepts the Android API and inter-component function calls of the plugin apps, modifies the parameters, forwards them to the Android system, then intercepts and relays the Android system responses back to the plugin apps, through the proxy. Meanwhile, the container app predefines the stub components and permissions to cater for those required by plugin apps, and it encapsulates plugin app components in stub components at the run time. In this way, multiple instances of the same app can bypass the UID restriction that disallows APKs with the same package name from having a different UID, and they can now run simultaneously [41]. To distinguish between the different guest applications, the host application assigns them different process IDs.

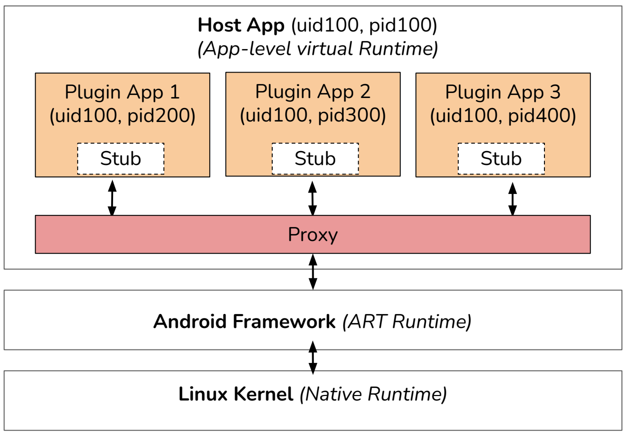

Host App (uid100, pid100)
(App-level virtual Runtime)
Plugin App 1 (uid100, pid200)
Stub
Plugin App 2 (uid100, pid300)
Stub
Plugin App 3 (uid100, pid400)
Stub
Proxy
Android Framework *(ART Runtime)*
Linux Kernel *(Native Runtime)*

Figure 2. App-level virtualization architecture with a container app managing the runtime of plugin apps in different processes but sharing the same unique User ID (UID).

The proxy layer heavily relies on hooking mechanisms to communicate between Android system services and plugin apps. For example, it hooks `ClassLoader` to load plugin apps' DEX files and the inter-application communication (IPC) to manage and maintain the lifecycle of the plugin apps' components (such as starting and stopping app activity). While implementations for Android app-level virtualization exist, this technique is still in its infancy, and its feasibility in an enterprise setting may vary between different implementations.

2.4. Anomaly Detection

Anomaly detection techniques are commonly used in automated log analysis [42–44], especially when dealing with high volumes of logs from different sources. During incident response, such techniques enable investigators and SOC analysts responding to incidents to detect anomalous behavior in systems by using machine learning algorithms to identify uncommon events or observations that raise suspicion, due to them differing significantly from the majority of events. Analysts select feature and time-based parameters within individual log entries that allow anomaly detection techniques to distinguish between log entries resulting from normal behavior and those resulting from potentially malicious, anomalous behavior. In cases where the content of log entries is verbose enough to include keywords that can serve as feature parameters, basic features can suffice to classify anomalies. However, in cases where the logs reflect normal behavior, derived features such as the frequency of the event may be more indicative of anomalous behavior. For instance, in the case of a messaging app hijack attack, features related to the contact name, message content, or the event itself may not be distinct enough to be considered anomalous (in the case of known contacts), given that this is the typical behavior of messaging apps. On the other hand, statistical or derived features related to an abnormal amount of messages may be more indicative of an anomaly. For instance, if the number of logs at a time is a time-based attribute, whereby if this value significantly differs (too high or too low) for a specific time data point, the data point is detected as possibly being anomalous. While selecting features is critical to the success of anomaly detection techniques, this still relies heavily on the content of the logs on which the anomaly detector operates and the known expected app behavior.

3. Motivation

3.1. Motivating Example

An employee, Bob, received a message on his company phone claiming his parcel had arrived and that he could track it by clicking the link. When he clicked the link, nothing happened, and he assumed the link was wrong. The link silently downloaded a malicious app that abused accessibility services intended to allow apps to interact with app GUIs on behalf of users, to grant access to a single permission so that the malicious app could hijack instant messaging (IM) apps installed on Bob's phone. The malware then propagated itself to all of Bob's contacts by hijacking the default messaging app's functionality to send the messages containing the malware link to Bob's contacts. It maintained stealth by automatically deleting the sent messages from Bob's app and hiding its icon from the home screen, so that it was no longer visible to Bob.

A day later, Bob received a text from his friend saying he had received a strange message from Bob. Bob did not see a copy of this message on his app and saw no new suspicious apps installed. However, he decided to alert the SOC at his company, which handled the incident response.

Bob's company demand that their employees have an EDR tool deployed on their phones that facilitates detection and response to attacks and onboarding of forensic sources on their enterprise SIEM (security incident event management, for example [45]) software technology (e.g., Splunk). The threat was not detected in real-time, as its stealthy nature and behavior signature did not match the EDR's known TTPs. Therefore, the SOC team initiated a forensic analysis of the logs produced by the EDR and found on their SIEM, to determine what happened on Bob's phone and if other employees were affected by it. However, logs from Bob's default messaging app and other sources only showed events and messages that Bob was already aware of. Messages sent and deleted by the malware were not found; therefore, no suspicious events could be detected.

3.2. Threat and System Model

This paper focuses on stealth attacks that aim to go undetected for longer by hijacking the legitimate benign app functionality of targeted sensitive apps, to perform attack steps. The victim is a general Android user with a stock Android device lured into installing a stealthy malware app that bypasses all existing protections and detection mechanisms in the Google Play Protect suite. This is not uncommon for Android malware, as recent studies have shown that 67% of malware found on phones was downloaded from the official Google Playstore [1,2].

The malware leverages attack vectors typically found on stock Android devices and apps to enable inter-app communication with sensitive apps, which the attacker can hijack to carry out attack steps.

Real World Attack Vectors and Examples

In recent years, several techniques have emerged that make for stealthier Android attack vectors. Android accessibility Trojans are a case in point [6,7]. Early instances [46] demonstrated how through phishing and the misuse of accessibility features, a malicious app could steal a victim's credentials and attack other benign apps and services by interacting with them, without the user's consent. This misuse has since shifted, from being leveraged to perform the actual attack, to being used to maintain stealth. Eventbot [9] and BlackRock [47] malware only request accessibility permission upon installation; the rest of the permissions required to perform the attack are obtained through the accessibility permission previously granted by the user. Even worse for the victim, the request for accessibility permission can be hidden from the user using UI confusion techniques, such as zero-permission tapjacking [16].

Malware developers can also exploit accessibility to hijack critical benign app functionality that coincides with the features they need, as seen in Figure 3. For instance, in the case of benign messaging app hijack attacks, attackers are interested in reading incoming messages (spying) or sending messages behind the victim's back (sending and deleting messages immediately). This functionality is not different from that typically offered by today's IM apps, other than that the initiator of these actions is a malicious actor, and the device owner is unaware of these events. This attack vector has been shown to enable stealthy living-off-the-land (LOtL) tactics [48], where key attack steps are delegated to benign apps, possibly only requiring the use of malware during an initial setup phase, to attain the maximum stealth [18]. Delegating an attack's core steps to benign apps has the consequence of bypassing malware detection mechanisms and making any follow-up threat detection and response more challenging, as reconstructing the attack steps distributed among trusted apps is not straightforward.

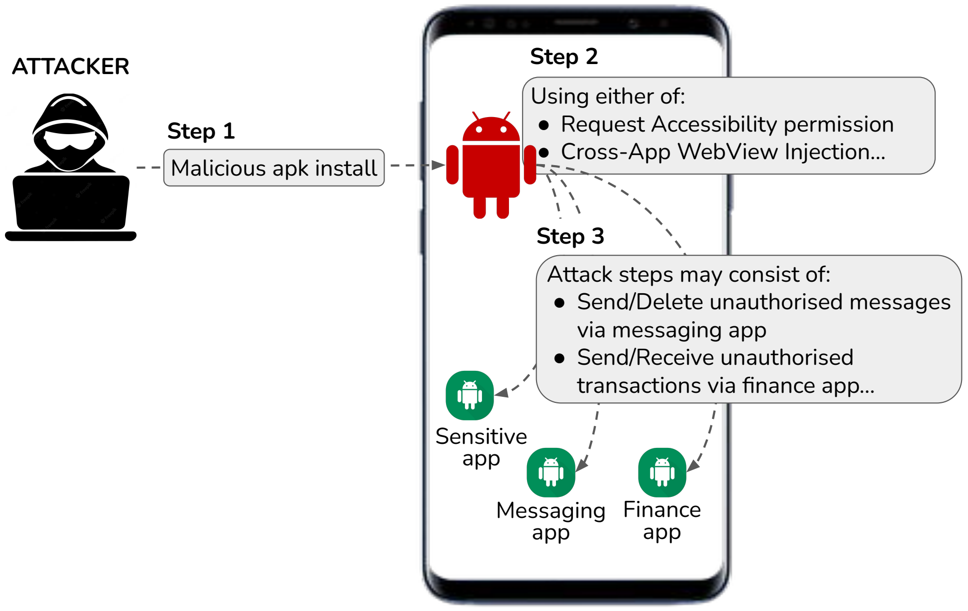

ATTACKER
Step 1
Malicious apk install
Step 2
Using either of:
• Request Accessibility permission
• Cross-App WebView Injection...
Step 3
Attack steps may consist of:
• Send/Delete unauthorised messages via messaging app
• Send/Receive unauthorised transactions via finance app...
Sensitive app
Messaging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

Figure 3. Malicious apps installed on the victim's device hijack benign app functionality to carry out attack steps and avoid detection.

Overall, any form of inter-app communication, whether for app functionality or testing purposes, can be similarly misused to avoid detection and complicate the incident response. Cross-app WebView infections (XAWI) [11], for instance, exploit legitimate cross-app WebView navigation, exposing the security risks of navigating an app's WebView through a URL. While a legitimate need for displaying the app's UI exists, to enable cross-app interactions, its abuse can lead to cross-app remote infection. In the case of messaging, malicious apps can misuse this functionality to send messages via another benign app. Another example vector is SMASHeD [49], which exploits the Android debug bridge. It enables malicious apps, requiring developer options to be enabled and requesting only the `INTERNET` permission, to read and write to multiple sensor data files at will, thus circumventing the Android sensor security model to stealthily sniff, as well as manipulate, many of Android's restricted sensors (even the touch input). PHYjacking [16] goes a step further and demonstrates how physical inputs used for authorization methods (e.g., fingerprint scanning) can also be hijacked through a threat model that exploits Android app implementation flaws found in 44% of 3000+ apps tested, as well as a powerful race-condition attack that can break the Android activity lifecycle model. Crucially, the threat model presented requires zero permissions, thus minimizing the malware component and bypassing permission-based detection mechanisms. Zygote and binder infection combined with rooting exploit [13], and third-party library infections [15] provide further attack vectors, potentially resulting in similarly stealthy attacks that render any efforts by classifier-based malware detectors futile. *Tap'n Ghost* [50] presents yet another relevant attack vector.

Another component of stealthy attack techniques is the limited forensic footprint that they leave behind, which has been demonstrated in previous work [17]. This means that when the victim eventually flags a strange behavior, incident responders have limited forensic sources available to reconstruct attack steps once the consequences take effect, possibly long after the attack occurred.

4. Proposed Solution

We propose VEDRANDO, an enhanced EDR for Android that accomplishes the challenging timely collection of volatile memory artefacts, along with the detection of a class of stealthy attacks that hijack benign apps, and which meets the requirements listed below:

R1 **Timely collection of app artefacts from memory.** The solution should include a special runtime to access app memory, thus being able to collect evidence of stealthy attacks that are not collected by state-of-the-art EDRs;

R2 **Extensible.** The techniques and technology enablers must create a generalized solution that works across multiple apps and attack scenarios, which would render the solution feasible to deploy;

R3 **Minimally Invasive.** The solution should be acceptable in an enterprise environment using stock Android devices and apps, thus not requiring device rooting or app repackaging and consequential reverse-engineering;

R4 **Detection of malware entry point and attack steps reconstruction.** Given the timely evidence collected from the memory, the solution should be able to reconstruct all the attack steps of a stealthy benign app hijack attack and detect the malware entry point using standard anomaly detection and correlation techniques.

Figure 4 gives an overview of the VEDRANDO architecture, which consists of two main components: the events collector and the attack detector. The events collector addresses the feasibility and implementation challenges of the timely collection of elusive evidence from memory, which discloses the attack steps of stealthy benign app hijack attacks (**R1**–**R3**) by extending the standard Android EDR with JIT-MF and leveraging app-level virtualization. The attack detector detects and reconstructs the attack steps of stealthy benign app hijack attacks (**R4**) through a detection methodology that applies standard anomaly detection and correlation techniques.

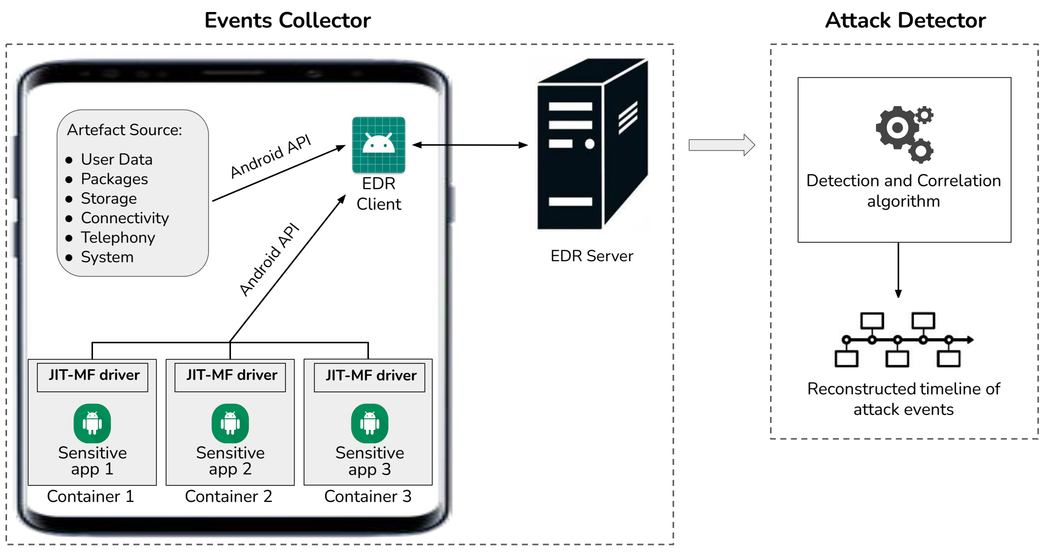


Figure 4. Overview of the VEDRANDO architecture.

4.1. Events Collector

The events collector component in Figure 4 illustrates a high-level view of our proposed memory forensics-enhanced EDR setup, comprising the following components: an EDR server, an EDR client (mobile app), and trusted app-level virtualization containers that each host a sensitive app that may be targeted by stealthy attacks, to hijack their functionality. While the makeup of each container is the same, different sensitive apps are hosted in different containers, to maintain the application sandbox protections that Android offers between apps out-of-the-box.

In the following sections, we describe how JIT-MF drivers were used to allow for the timely collection of artefacts from memory (**R1**) that can contribute to the stealth attack steps of a benign app hijack, while ensuring extensibility (**R2**) by moving away from app-specific JIT-MF drivers that render the technique infeasible at a large scale. We also describe how our solution leverages app-level virtualization, to remove the need for app repackaging, yet still functions on stock Android devices (**R3**). Finally, we illustrate and describe the complete setup of the enhanced Android EDR, along with implementation considerations.

4.1.1. Infrastructure-Centric JIT-MF Drivers

We recall research in previous work [25] that laid the groundwork for the feasibility of JIT-MF driver development by addressing the limitation that requires JIT-MF drivers to be specific to the targeted app and attack scenario at hand. This limitation meant that JIT-MF driver development required app reverse-engineering, which rendered the development process of JIT-MF driver development impractical.

The JIT-MF driver development process must be practical to ensure our solution is feasible (addressing **R2**). Infrastructure-centric JIT-MF drivers render the JIT-MF driver development process feasible by ensuring that a single JIT-MF driver can remain relevant across app versions and stay functional across different applications. The overarching idea of this type of JIT-MF driver calls for a modified driver development approach that leverages the common subset of the applications' codebase that interacts with commonly-used infrastructure, rather than a application-specific codebase, for *Trigger_points* and *Evidence_objects* selection. As shown in Figure 5, the underlying infrastructure is generally more stable and widespread across different applications and versions, allowing infrastructure-based JIT-MF drivers to remain usable across different applications and versions (unlike application-specific drivers, which need to be developed from scratch for every application and version). A crucial step for infrastructure-centric JIT-MF driver development involves identifying the key application events that may be hijacked and the commonly used, readily-available infrastructure that enables these events [25].

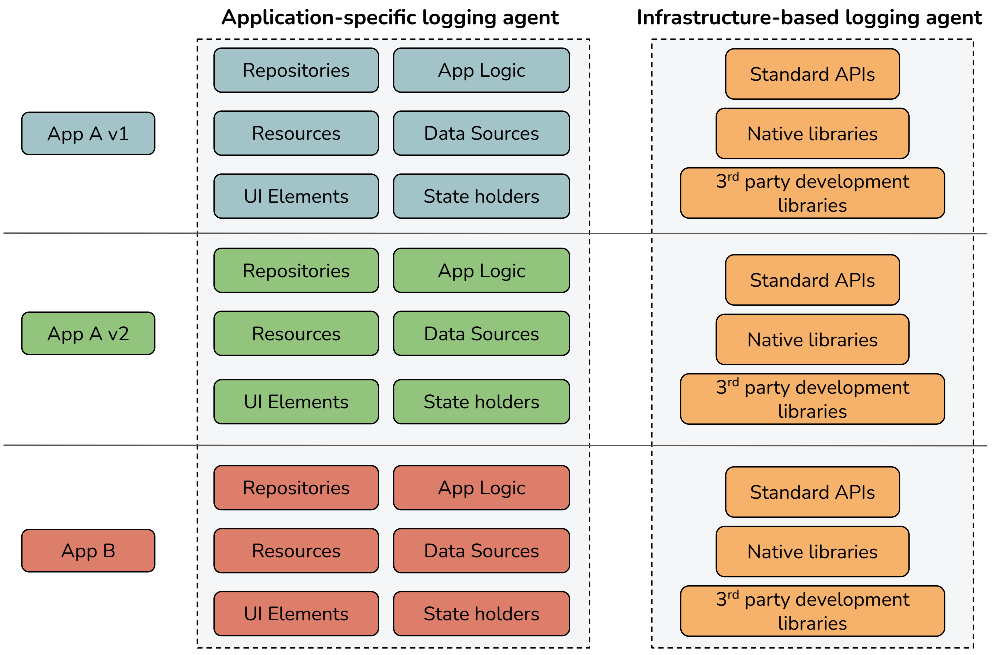


Figure 5. JIT-MF drivers leveraging application-specific codebase versus JIT-MF drivers leveraging common underlying infrastructure-interfacing code, which is expected to be more stable across versions (App A v1, v2) and applications (App A, B). Different color codes signify application-specific codebases, whereas the same-colored codebases reflect common APIs across applications and versions [25].

4.1.2. Android App-Level Virtualization

We extended the VirtualApp framework [40] to develop an enhanced container that collects artefacts in a timely manner from the memory of plugin apps that run inside it. The new VirtualApp container contains an additional library, to serve as the JIT-MF driver runtime, implemented using Frida's Gadget shared library (https://frida.re/docs/gadget/ accessed on 30 June 2023), and JIT-MF drivers are implemented as Javascript code interpreted by that library. Sensitive stock Android apps that require monitoring and logging of artefacts from memory due to the potential for hijack are placed inside external storage and picked up by the VirtualApp container to be installed as plugin apps. The JIT-MF driver runtime is loaded when the plugin app starts, which enables the timely collection of artefacts from the plugin app memory (addressing **R1**) without requiring app repackaging, thus also addressing **R3**.

4.1.3. Working Prototype

We implemented the events collector component of VEDRANDO by extending ReLF [27], the only open-source EDR tool available for mobile phones to the best of our knowledge, with the ability to collect critical evidence of sensitive app events found in memory produced by JIT-MF drivers. ReLF extends GRR [51], an open-source, scalable system developed by Google for remote live forensics and incident response and enables forensic investigations of Android devices by acquiring various forensic artefacts from devices (as many as any other such forensic tools, see Table 1). As with typical EDRs, the setup involves having ReLF clients on mobile phones, from which events are collected and sent to a ReLF server. The ReLF client may be built and deployed as a user or system app. The latter has access to more forensic sources (see sources marked with ∗ in Table 1) but requires root access. ReLF client apps built as user apps interact with the underlying system through Android APIs or the low-level ReLF native service using inter-process communication (IPC) [27]. JIT-MF drivers in different containers may be the same if the sensitive apps (1, 2, and 3 in Figure 4) use a common infrastructure (which the evaluation results demonstrate is very likely the case). In the specific case of our working prototype, the EDR client and server were the ReLF client app (built as a user app to comply with **R3**) and server, respectively.

Artefact Collection

While the app is in use, JIT-MF logs are populated continuously with *Evidence_objects* from memory, upon the invocation of the specified *Trigger_points* in the JIT-MF driver of the container. When the alarm is raised, the ReLF server can invoke artefact collection flows, instructing the ReLF client to collect any pending logs not yet collected through continuous monitoring, to be sent back to the server as part of evidence collection to aid the ongoing investigation. As shown in Figure 4, the ReLF client leverages the Android API to collect all Android forensic sources, including logs containing the in-memory evidence collected by the JIT-MF driver deployed within VirtualApp. For logs generated by JIT-MF drivers containing evidence from app memory, the client uses the Android API to search for files on the device with a `*.jitmflog` extension.

Other Implementation Considerations

For the prototype described above, the JIT-MF driver and logs generated are placed in the temporary directory (`/data/local/tmp`) and external storage (`/storage/emulated/0`), respectively, to enable ease of automation. Furthermore, we assume the container can be trusted [52].

EDR tools (including the events collector component of VEDRANDO) deployed in enterprise settings must comply with standard security measures for which existing implementation solutions exist. Therefore, in a realistic environment, scoped storage would need to be used to appropriately store JIT-MF logs and drivers, thus ensuring secure access to these critical contents. If these were to fall prey to a malicious actor, then critical evidence might be lost or remain uncollected. Similarly, to verify JIT-MF drivers, digital signatures can be used and approved by the device owner, app developer, device manufacturer, or a combination thereof. This would significantly reduce the threat of deploying malicious JIT-MF drivers and safeguard against any privacy concerns of the device owner. All sensitive apps being monitored (plugin apps) should be automatically updated with the latest changes published by app vendors. In so doing, the events collector avoids the need for re-installation/sign-up. Furthermore, they should retain the security features provided by Android out-of-the-box, mainly that only authorized access to the app should be allowed.

4.2. Attack Detector

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

The attack detector component of VEDRANDO comprises the detection algorithm outlined in Algorithm 1. The algorithm uses an existing, standard anomaly detection method to detect anomalies in the JIT-MF logs, then correlates anomalies with events collected from other logs found on the device, to reconstruct all the attack steps, including the malware entry point (addressing **R4**). The algorithm takes as input the logs produced by the events collector component, comprising JIT-MF logs with evidence objects from app memory and other logs found on the device (see Table 1), and a user-defined configuration *Config c*. The configuration variable *Config c* holds settings related to generating the anomaly detection model. Namely, it comprises: (i) the anomaly detection method (a); (ii) associated features selected ($[f_1 \ldots f_n]$); (iii) the anomaly threshold value t, which will be used to identify data points as anomalous; and (iv) a list of app-specific regex keywords ($[p_1 \ldots p_n]$) used during the correlation of events. The algorithm outputs a list of events *Correlated_Events e* attributed to the complete attack steps.

4.2.1. Anomaly Detection

All the entries from different log sources are parsed (*line 1* in Algorithm 1), so each entry has three main fields: (i) timestamp, (ii) log source, and (iii) activity. JIT-MF logs are filtered to remove duplicates. Furthermore, in the case of both the third-party app and JIT-MF logs, we further filter the logs so that only sources of evidence related to the evidence object are considered. For instance, in a messaging hijack attack, where the evidence object is a message sent from the user's phone, the evidence collected from the app is its database, comprising many tables and possibly also containing data unrelated to messaging, e.g., app themes, which may cloud the investigation. The function *GetAnomalies*() is then called, with the following parameters: (i) parsed JIT-MF logs (J); (ii) logs from other sources (O); and (iii) configuration settings (*Config c*).

The function *GetAnomalies*() first generates a machine learning anomaly detection model based on the machine learning method and features defined in the user-inputted configuration (*line 4*). The model m is then applied on the parsed and filtered set of JIT-MF logs J using the threshold defined in the configuration settings (*line 5*). The anomalous JIT-MF log entries revealed by the anomaly detection model are considered anomalous JIT-MF events. These are then correlated (*line 6*) with other log events (JIT-MF logs and logs from other sources) using the app-specific correlation regex keywords given parameter ($[p_1 \ldots p_n]$). The function returns the result of the *Correlate*() function.

Algorithm 1 Anomaly detection and correlation algorithm

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

 1: $J, O \leftarrow ParseLogs(J, O)$
 2: $J, O \leftarrow GetAnomalies(J, O, c)$

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

4.2.2. Correlation

JIT-MF log events include a timestamp and metadata of the *Evidence_object* definition as described in the JIT-MF driver. Regardless of the driver implementation or the app, the contents of the *Evidence_object* can be parsed to derive relevant keywords used during the attack step. App-specific correlation keyword regex retrieves the relevant keywords from anomalous JIT-MF log entries. The *Evidence_object*'s makeup is app-specific; therefore, the regex pattern used to retrieve this metadata or identifier from a log entry must also be app-specific. That said, there are cases where the keyword regex is the same across apps, due to formatting standards, e.g., the object ID may be in UUID format, which is a standard format.

The *Correlate*() function accepts as input the anomalies detected, JIT-MF logs, logs from other sources, and the app-specific correlation regex keywords. If the anomaly is a timestamp (as is the case with time-based anomaly detection), JIT-MF logs at that time are retrieved (*lines 12–17* in Algorithm 1). The *Evidence_object* of the anomalous JIT-MF log entry is retrieved (*lines 18*) and used to perform correlation, as follows: The algorithm correlates anomalous JIT-MF log events with events in other log sources, based on two mechanisms: (i) feature-based correlation and (ii) time-based correlation. It is unlikely that normal events have identical keywords in their *Evidence_object*. However, in the case of malware, especially during propagation, *Evidence_object*s containing matching keywords are expected. Therefore, events in other log sources that contain identical keywords to those found in anomalous JIT-MF log events (*lines 21–26*) are considered further attack steps, and are added to the list of *Correlated_Events e*. This is referred to as feature-based correlation. Any other attack steps performed in the attack are assumed to have happened in the period within which the correlated list of attack steps occurred. Therefore time-based correlation is used to search for other events that occurred in other logs when the JIT-MF log anomalies were detected (*lines 30–33*). This ensures any attack steps carried outside the app functionality are also disclosed. Any log entries found through correlation are entered into a set of correlated events and returned to the analyst or investigator as the complete list of the attack steps carried out.

5. Experimental Evaluation and Results

We evaluated the feasibility of VEDRANDO's events collector component based on the JIT-MF driver development effort required and the compatibility with app-level virtualization. Experiments supporting this evaluation involved (i) carrying out a coverage analysis of the most popular 550 apps (this is the maximum number of apps returned by AppBrain statistics) on Google Playstore, to find the most commonly used underlying infrastructure libraries (using statistics obtained from AppBrain [26]) that can be leveraged for JIT-MF driver development (Section 5.1), and (ii) executing popular apps on Google Playstore within the events collector setup, to evaluate their compatibility with VirtualApp containers equipped with infrastructure-based JIT-MF drivers and to determine the introduced runtime overhead (Section 5.2). The apps considered in these experiments spanned more than 39 categories, including the messaging and finance categories, which are the primary targets for damaging stealth attacks [53] and for which VEDRANDO could be a solution.

Finally, we evaluated the effectiveness of VEDRANDO's attack detector component by simulating ten stealthy instant messaging (IM) hijack case studies, targeting ten of the most popular Android IM apps. Our results showed that, given timely captured evidence from memory collected by VEDRANDO's events collector, anomaly detection and correlation techniques could be used to reconstruct all attack steps of the stealthy benign messaging hijack attacks targeting popular Android messaging apps.

5.1. JIT-MF Driver Setup

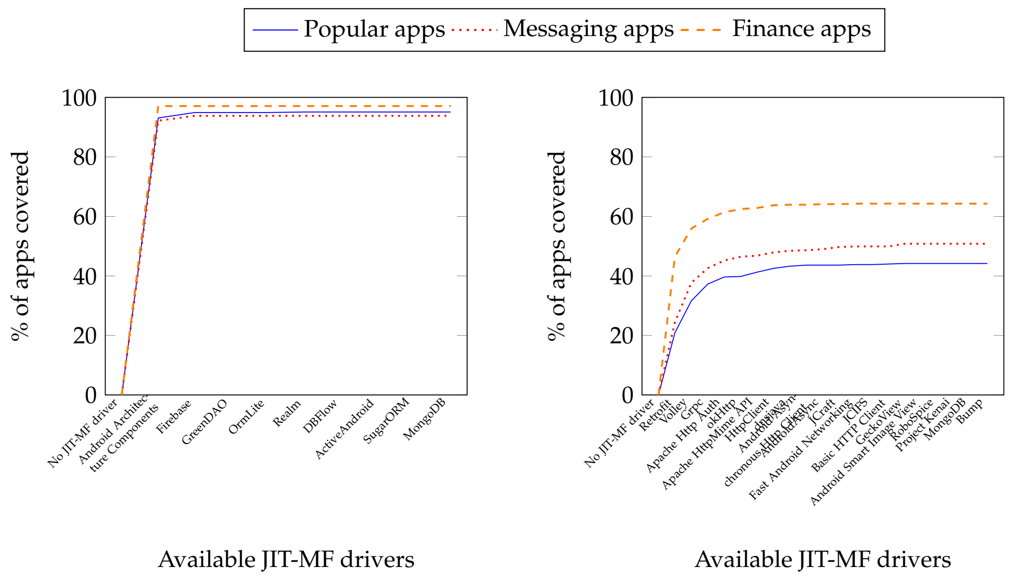
This analysis aimed to identify the infrastructure handling core app functionality (see Section 4.1.1) that is commonly used among popular apps in the general, messaging, and finance categories, enabling a more feasible, generic JIT-MF driver that can be used across apps and app versions.

We *extended existing results* [25] and considered the data provided by AppBrain, a service that provides statistics on the Android application ecosystem, including library adoption by different apps in different categories. AppBrain categorizes libraries used in Android applications using tags, depending on the functionality provided by the library. Out of the 41 possible categories, we identified the database (storage) and network libraries as critical infrastructures that typically handle data in sensitive events. Database functionality allows data to be stored and retrieved on the devices where the app is installed, and network functionality handles the data to be transferred over the network.

Coverage Analysis

Figure 6 shows the usage distribution of the database and network libraries by the 550 most popular apps. The percentage of apps covered suggests that database libraries are more widely adopted than network libraries across all app categories. The graphs show that, overall, usage of database libraries is common across 93.8% and 97.1% of messaging and finance apps, respectively, and 95.1% of all apps. Whereas network libraries are much less prominent, being adopted in only 50.8% and 64.3% of messaging and finance apps, respectively, using the most popular network libraries. The figure is even lower, 44.2%, for popular apps in general.

Furthermore, the database library usage graph shows a much steeper incline, meaning that a large number of apps use the same small number of database libraries. Specifically, the most widely adopted database infrastructure was Android architecture components, with 93.1% and 97.1% adoption among the most popular 550 messaging and finance apps, respectively, and adopted among 92.2% of all apps. At its most native level (see Section 4.1.1), Android architecture components refers to storage management through an SQLite Database (https://developer.android.com/training/data-storage/sqlite accessed on 30 June 2023). While keeping in mind that these values were obtained via static analysis of apps, this still bodes well for the extensibility of JIT-MF drivers and the feasibility of JIT-MF driver development, since one SQLite-based JIT-MF driver could potentially be successful on an extensive range of apps.



(**a**) The number of storage infrastructure-based JIT-MF drivers needed to forensically enhance the % of apps.

(**b**) The number of network infrastructure-based JIT-MF drivers needed to forensically enhance the % of apps.

Figure 6. Storage and network library adoption by the 550 most popular apps in February 2023 [26].

5.2. Runtime Evaluation

We evaluated the feasibility of VEDRANDO's events collector VirtualApp and JIT-MF driver setup, in terms of its compatibility with Android apps and the resulting performance overheads. To do this, we selected a set of apps from the 100 most popular apps in Google PlayStore in February 2022 (as listed on AppBrain), which had not previously been installed on the phone and were not manufacturer-specific, resulting in a total of 33 apps.

A stock (unrooted) Google Pixel 3a physical phone, with eight processors and 4 GB RAM, was used, which runs on arm64-v8a CPU architecture and Android version 9 (as required by VirtualApp). The apps selected were downloaded from APKPure (https://apkpure.com/ accessed on 30 October 2022) using *apkeep* (https://github.com/EFForg/apkeep accessed on 30 October 2022) to ensure that the APKs downloaded complied with the architecture and Android version. We used the *UI Exerciser Monkey* tool (https://developer.android.com/studio/test/other-testing-tools/monkey accessed on 30 October 2022) to exercise each app's functionality by injecting 20 random UI events with a throttle of 30 s, which allowed the virtual environment to spawn the app, but which could not be reset to execute the rest of the events due to limitations of *UI Exerciser Monkey*. A seed value was used to ensure that the same app events could be repeated in the case of multiple runs.

We installed and executed the 33 apps directly on the device, to check typical resource usage. We ran the apps in a standard VirtualApp container, to evaluate their compatibility with the virtual environment. In this step, we collected the overhead introduced by the Android virtualization regarding CPU and memory usage. At the end of this phase, we identified five apps that triggered an exception due to incompatibility with the virtual environment. Thus, we discarded such apps from the rest of the experiments. The remaining 28 apps were executed three times: (i) directly on the device, (ii) inside a simple VirtualApp container, and (iii) inside a VirtualApp container equipped with an SQLite-based JIT-MF driver (as implemented in VEDRANDO's *Events Collector*). The results were averaged over ten runs.

Results

Table 2 shows the minimum, average, and maximum overhead values expressed in percentage points (pp). In the first column, we compared the execution of apps in a plain VirtualApp with the traditional execution method (no virtualization). We computed the overhead for the VirtualApp container as implemented in VEDRANDO's events collector component (i.e., second column) compared to the execution in a plain VirtualApp environment. Since the execution of an app under virtualization is composed of two processes (the container and plugin), the overall amount of CPU and memory is given by the sum of the overhead of these two processes.

Table 2. Overall CPU and memory usage overheads in percentage points (pp) for the 28 most popular apps, when executed within VirtualApp and a JIT-MF-enhanced version of VirtualApp, respectively.

		VirtualApp ***(Added pp Overheads on Device)***	**VirtualApp with JIT-MF** ***(Added pp Overheads on Plain VirtualApp)***
CPU	min.	−0.46	+1.2
	avg.	+2.83	+2.06
	max.	+2.76	+6.79
Memory	min.	−0.48	+0.23
	avg.	+0.56	+0.18
	max.	+1.19	+0.29

The results from Table 2 show that when introducing virtualization through VirtualApp, there was an average increase of 2.83 pp in CPU usage and 0.56 pp in memory usage. The results for the container as implemented in the events collector component, using an SQLite-based JIT-MF driver, show that the additional average overhead introduced was negligible, i.e., an increase of 2.06 pp for the CPU usage and 0.18 pp for the memory. We concluded that this increase was caused by the overhead required by JIT-MF drivers to capture memory dumps for trigger points performed through instrumenting methods. The overall additional CPU usage incurred by VEDRANDO's *events collector* component when using SQLite-based JIT-MF drivers was on average 4.89 pp, rendering it feasible in terms of runtime performance in a real-world scenario. This, however, may vary depending on the type of JIT-MF driver used in the VirtualApp container.

5.3. Attack Investigation Case Studies

We evaluated the effectiveness of VEDRANDO's attack detector by measuring its ability to reveal attack steps related to stealthy benign app hijack attacks in a realistic scenario. Rather than assessing the performance of existing anomaly detection models on a large dataset, we aimed to demonstrate how the anomaly detection methods typically available to SOC analysts through their SIEM setup can be used to detect anomalous events related to benign app hijack attacks when provided with JIT-MF logs produced by VEDRANDO's events collector component. To this end, similarly to other related works [54–56], we presented a qualitative case study for a benign instant messaging (IM) hijack attack targeting Android's ten most popular IM apps, as shown in Table 3, following the threat model described in Section 3.2.

Table 3. List of applications used in the case study.

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

In Sections 5.3.1 and 5.3.2, we describe the case study setup and the settings used for the proposed anomaly detection and correlation algorithm (Algorithm 1), respectively. Section 5.3.3 shows the summarized results concerning the reconstructed attack steps obtained for the ten case studies.

5.3.1. Case Study Setup

Figure 7 shows the experiment setup and flow, comprising an implementation of the working prototype for VEDRANDO, shown previously in Figure 4, and the investigation flow indicated by arrows. A stock (unrooted) Google Pixel 3a physical phone was used, on which an implementation of VEDRANDO's events collector component was deployed, using an SQLite-based JIT-MF driver (https://gitlab.com/bellj/vedrando/-/tree/main/sqlite-jitmf-driver.js), given that the popularity of this infrastructure among messaging apps has already been established (see Figure 6a). Normal traffic on each app consisted of loading and sending instant messages. This was simulated using *AndroidViewClient* (https://github.com/dtmilano/AndroidViewClient accessed on 30 October 2022), assuming that the user messages random contacts from his list of contacts, waiting a random amount of seconds (between one and ten) before sending the message. We acknowledge that this simulation of normal traffic may be a threat to validity. However, we claim that the simulated traffic generated within the case study time window was a sufficiently realistic representation to provide a basis for our study.

Benign IM App Hijack Simulation

Following the threat model outlined in Section 3.2, we simulated an IM hijack attack scenario that misused IM functionality for propagation (step 1 in Figure 7), akin to the popular Flubot [8] malware, using additional stealth measures to conceal attack steps. The simulated attack steps were carried out using `adb` shell commands and *AndroidView-Client* https://github.com/dtmilano/AndroidViewClient, relying on the attack vectors, as described in Section 3.2.

Table 4 shows the ground truth timeline of events carried out to simulate the attack scenario for this case study. A malicious message is received containing a link to a malicious app (Table 4 ❷). Once the user clicks on the link, the app (a fake app called *demo.apk*) is silently installed (Table 4 ❸) and propagates to the user's contacts via the default IM app installed (in this case, the apps in Table 3). To attain stealth, the simulated attack deletes the sent messages from the victim's phone (Table 4 ❺) and hides by removing the malicious app icon from the home screen so that the victim is unaware of the malicious app and it goes unnoticed by the victim for longer. ❻ in Table 4 is a *Trigger Event*; that is, it alerts the user that a suspicious event has possibly occurred, which initiates an investigation process. It is typical for realistic malware aiming to be stealthy to wait until it is the right time to execute [57]. In this case, the malware waits until the hijacked app is not in use, so as not to alert the user of abnormal behavior. Due to these stealth measures and additional ones that the malware uses to hide its attack steps, the trigger event occurs long after (in this case, almost an hour later) the attack, meaning that the malware would have hidden its tracks, leading to delayed detection.

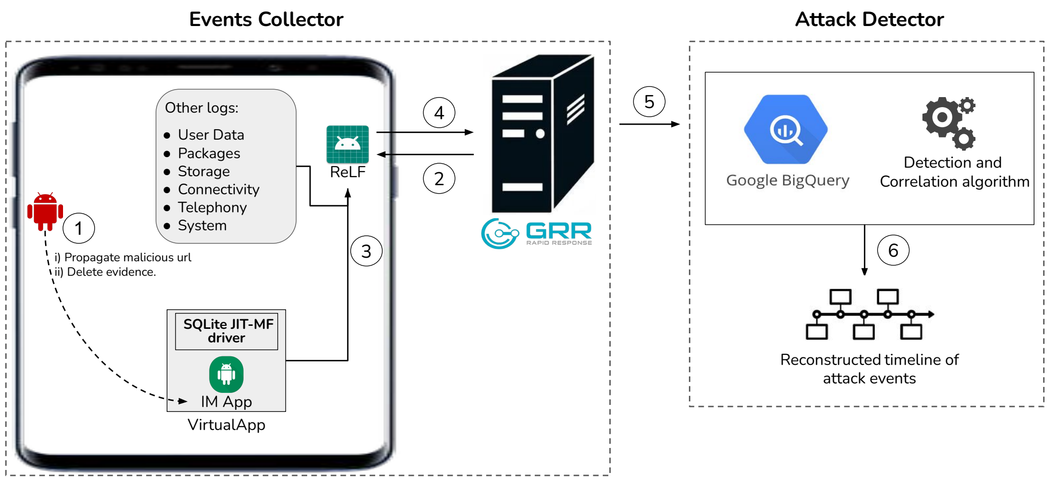


Figure 7. Complete case study experimentation flow.

Investigation Setup

We assumed the role of an SOC analyst in an enterprise and started an investigation process by invoking commands from the GRR ReLf server (step 2 in Figure 7) to collect evidence artefacts from the victim's phone, including JIT-MF logs produced by the JIT-MF driver (step 3). SOC analysts are typically equipped with SIEM services that provide access to out-of-the-box anomaly detection tools. The GRR ReLF server has bindings to Google BigQuery (https://cloud.google.com/bigquery accessed on 15 April 2023), a service that enables scalable analysis over petabytes of data and provides machine learning capabilities including anomaly detection, which we use as a SIEM equivalent. During the investigation procedure followed in this evaluation, the artefacts collected by the ReLF client were sent back to the GRR ReLF server (step 4) and saved in Google BigQuery datasets (step 5). VEDRANDO's attack detector detection and correlation algorithm used Google BigQuery's machine learning API to detect anomalies in the collected JIT-MF logs. These anomalies were then correlated to events from other logs, to reconstruct the attack steps executed by the stealth benign messaging app hijack attack (step 6).

Table 4. Ground truth timeline of events for this case study.

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

5.3.2. Detection and Correlation Configuration

The investigation procedure outlined above was carried out after the attack hijack scenario was executed on each targeted messaging app shown in Table 3. Once the logs for each case study had been retrieved, we implemented and executed the detection and correlation algorithm (Algorithm 1) using the configuration described below.

Anomaly Detection Models

Google BigQuery ML [58] provides anomaly detection capabilities through four machine learning model types: ARIMA_PLUS, K-means, PCA, and Autoencoder. All these models are unsupervised and can therefore detect anomalies without needing labeled data. ARIMA_PLUS detects anomalies in time series data, while the others detect anomalies in independent and identically distributed random variables. For our evaluation, we used these models with selected applicable parameters and features as configuration input to our detection algorithm (Algorithm 1) to measure the algorithm's effectiveness in detecting and reconstructing attack steps. All collected logs (including JIT-MF logs) were processed in BigQuery, and the preprocessed log content was used for building the different models. Hyperparameter tuning is commonly used to improve model performance, by searching for optimal hyperparameters. During our evaluation, we used the default and recommended Vertex AI Vizier algorithm to tune the hyperparameters https://cloud.google.com/bigquery/docs/reference/standard-sql/bigqueryml-hyperparameter-tuning.

Dataset

The evaluation of machine learning algorithms typically involves using large, established datasets. However, this evaluation aimed to demonstrate the value that JIT-MF logs bring to the incident response process, by showing that evidence in these logs enables existing machine-learning anomaly detection models to detect anomalies in benign app activity related to an app hijack, which can help reconstruct attack steps. Therefore, the dataset used to train the anomaly detection models in our evaluation was similar to what an SOC would have available in such an incident. This comprised logs typically collected by EDRs (shown in Table 1) and JIT-MF logs that were populated during the case study (which involved both the attack and normal traffic) and collected as part of the investigation process by VEDRANDO's events collector component.

The VirtualApp container used by the event collector was built and deployed to the phone in debug mode, and therefore its app data could be retrieved. VirtualApp app data houses the data produced by plugin apps, and therefore relevant third-party app forensic sources could also be accessed and collected as forensic sources. When working with a

VirtualApp container app that is not running in debug mode, forensic analysts can opt to use app features such as "backup" or collaborate with the device owner to collect the evidence that is present in the app. Once the sources were collected, relevant data were extracted related to the app's main functionality (in this case, messaging), converted into logs, and transferred to our Google BigQuery dataset.

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

Features

Table 5 shows the features used per anomaly detection method to generate the anomaly detection models during the execution of Algorithm 1. Features were selected based on the anomaly detection method and the knowledge that JIT-MF logs may contain evidence of offloaded attack steps that are not visible in other forensic sources. Log entries from multiple sources were parsed, so that each had a timestamp, forensic source, and activity. However, the format of the content inside the activity field differed from one forensic source to another, both across sources and in the case of app-specific logs and JIT-MF logs, and even across apps. Rather than parsing each log type individually for each app and forensic source, we used derived features, in the form of log entry amounts per feature grouped by a time window.

Table 5. List of features used for anomaly detection model generation, as implemented in Algorithm 1. The time units used for each model generated are described in Table 6.

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

Feature 1 represents the discrepancy in the log entry amount between that produced by all forensic sources (excluding JIT-MF logs) and the amount found in JIT-MF logs.
Feature 2 represents the total amount of log entries collected from all sources of the events collector component.
Features 3 represents the log entry amount collected from JIT-MF logs.
Features 4–9 represent the log entry amounts collected from each distinct forensic source (excluding JIT-MF). While Table 1 shows that a typical collection involves retrieving multiple forensic sources, only five were populated during the case study (e.g., no connectivity data were reported).
Features 10–14 represent the log entry amounts collected from JIT-MF logs with distinct SQL statements. Since the JIT-MF logs in these case studies were generated using an SQLite JIT-MF-based driver, log entries included SQL statements which process the message object (as shown in Listing 2). The `SELECT`, `INSERT`, `REPLACE`, `UPDATE` and `DELETE` SQL statements

can be considered to reflect app functionality related to the processing of the message object. We considered the amount of logs containing a specific SQL statement a feature.

The models in Table 6 were generated based on the features described. Two models were created for each combination of anomaly detection methods and features or feature sets (in the case of K-means, PCA, and Autoencoder). In the case of Google BigQuery's ARIMA_PLUS, log entries were automatically grouped in sixty-second time windows. For ARIMA_PLUS we used two different feature normalization properties (Standard Scaler and Min Max Scaler (https://cloud.google.com/bigquery/docs/reference/standard-sql/bigqueryml-preprocessing-functions), whereas for K-means, PCA, and Autoencoder we aggregated and counted events every thirty (30s) and sixty (60s) seconds. Each model in Table 6 was created for every targeted app in the case study.

Listing 2. JIT-MF log entry sample containing SQL statements, generated while using WhatsApp, Telegram, and Signal Android apps using a SQLite JIT-MF-based driver. Other metadata that did not contribute to the investigation were redacted. However, the full content of the artefacts can be found in the accompanying paper repository (https://gitlab.com/bellj/vedrando/-/tree/main/forensic_artefacts_collected).

```
{"time": "1681643999", "event": "Telegram Message Sent", "trigger_point(s)": "sqlite", "object": {"REPLACE INTO
      messages\_v2 VALUES(19037, 961166549,..., 1676821892, n8<J'QY<J9xcRHey, I think something is wrong with your
      phone. You sent me a suspicious message.,...)"}}
{"time": "1676928079", "event": "Whatsapp Message Sent", "trigger_point(s)": "sqlite", "object": {"INSERT INTO
      message(...,sender_jid_row_id,...receipt_server_timestamp,text_data,...) VALUES
      (...4,18446744073709552000,...,18446744073709552000,DHL: Your parcel is arriving, track here: https://
      flexisales.com/dhl1eep7j88cc5z3,...)"}}
{"time": "1678038113", "event": "Signal Message Sent", "trigger_point(s)": "sqlite", "object": {"INSERT INTO message(
      view_once,receipt_timestamp,..,body..,recipient_id) VALUES (0,18446744073709552000,...,DHL: Your parcel is
      arriving, track here: https://flexisales.com/dhl18446744073709552000eep7j88cc5z3v,...,4)"}}
```

Table 6. Models generated based on the selected features.

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

Anomalies were detected depending on the model used and the threshold set. ARIMA_PLUS is a univariate time-series model that uses a single feature to detect anomalous data points across historical data. In contrast, the K-means, PCA, and Autoencoder models use multiple features for clustering (K-means) and dimensionality reduction (PCA, Autoencoder), which results in the identification of anomalies based on outliers and reconstruction loss. Each model supports a custom threshold for anomaly detection in Google BigQuery ML (https://cloud.google.com/blog/products/data-analytics/bigquery-ml-unsupervised-anomaly-detection). For ARIMA_PLUS models anomalies are identified based on the confidence interval for that timestamp. If the probability that the data point at that timestamp occurs outside of the prediction interval exceeds a given probability threshold, the data point is identified as an anomaly. Furthermore, since Google BigQuery returns the feature value, our detection algorithm implementation also checked that, for the given anomaly found, *Feature 1* (the discrepancy between logs) was greater than 0. For the other models, anomalies were identified based on the value of each input data point's normalized distance to its nearest cluster. The data point was identified as an anomaly if that distance exceeded a threshold determined by the given contamination value. The contamination value defined the proportion of anomalies in the training dataset. This value ranged from 0.1 to 0.5, where 0.1 and 0.5 mean that 10% and 50% of the training data used to create the input model, respectively, were anomalous. Whereas for the ARIMA_PLUS models, a lower threshold value made the data points more likely to be considered anomalous, for the other models, a larger contamination value (threshold) made the data points more likely to be considered anomalous.

High-Level Event Reconstruction and Correlation

Before executing the attack detector component of VEDRANDO, we conducted a preliminary manual analysis of the logs generated by the SQLite JIT-MF drivers, to determine how low-level JIT-MF log entries could be combined to form more indicative high-level events. This analysis revealed that, in the case of SQLite-based JIT-MF drivers and the apps used in the case studies, a regex pattern for JIT-MF logs generated by each app could identify a log entry that reflected the actions of several JIT-MF low-level events generated after an action had occurred. For instance, when a message is sent (high-level event), multiple JIT-MF log entries (low-level events) are generated (related to updates made to several tables in the database). A single entry, however, is identified as explicitly updating and inserting content into the app's specific `messages` table in the database. This manual process was also required to select the correlation regex keyword specific to each app. In these case studies, keyword regex aimed to extract the message content and identifier (ID). Therefore, we defined regex string patterns for these two keywords for each app, so that any message content or message ID found in the log entries could be correlated with related events.

5.3.3. Attack Investigation Results

For each targeted app, the ground truth attack steps of the simulated benign app hijack were recorded as shown in Table 7 (a subset of the events shown in Table 4). We demonstrated the value of timely evidence collected from the app memory for each attack by first showing that, upon manual inspection, evidence related to the attack steps was predominantly only collected by VEDRANDO's events collector. Furthermore, based on the detection methodology described in Algorithm 1, we showed in the realistic context of an ongoing investigation that the evidence in JIT-MF logs was critical for detecting and responding to anomalies.

Artefacts Recovered by JIT-MF Logs

Table 7 summarizes which critical attack steps executed on all targeted messaging apps were found in logs typically collected by an EDR and in those collected by VEDRANDO's events collector component that included JIT-MF logs.

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

Reconstruction of Attack Steps

Now that we have established that only JIT-MF uncovered these anomalies, we focus on the thresholds and model parameter selection that performed best with the attack detector to uncover these anomalies. Tables 8 and 9 show the effectiveness of the detection and correlation algorithm in VEDRANDO's attack detector component for reconstructing stealthy attack steps executed during the stealth IM hijack case studies, with the input parameters defined in Section 5.3.2. For each model and threshold input combination, we calculated the average recall, precision, and F1-scores, to measure the overall accuracy of the reconstructed set of events returned by the attack detector when compared to the ground truth set of events executed by the attack. The F1-score combines precision and recall values. Therefore, the higher the F1-Score, the more accurate the list of attack steps returned by VEDRANDO's attack detector.

Table 7. Comparison of the ground truth attack steps disclosed and detected by a typical EDR and by VEDRANDO (columns 3 and 4, 5 and 6), respectively.

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

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

Table 8. Table showing the average F1-scores for the reconstructed attack steps across all case studies, generated by the combined anomaly detection and correlation algorithm when using ARIMA_PLUS models, with varying threshold values. The threshold values in this case are inversely proportional to the allowance for anomaly probability.

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

Table 9. Table showing the average F1-scores for the reconstructed attack steps across all case studies generated by the combined anomaly detection and correlation algorithm when using K-means, PCA, and Autoencoder models, with varying threshold values. The threshold values, in this case, are proportional to the allowance for anomaly probability.

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

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

Further analysis of the results obtained by these three models revealed that the average recall value across apps increased at a faster rate than the precision value decreased. This was because, for individual case studies (which varied depending on the model used), a more lenient threshold value was required to obtain the same recall value that the other apps obtained with less lenient threshold values. Figure 8 shows this for the specific case of the model input parameter resulting in the best overall F1-score value (M7). In this case, 90% of the apps used in the case studies reached an average 100% recall value on the reconstructed set of attack steps when the threshold was set to 0.3. However, the WhatsApp Business attack steps were only detected as anomalies when the threshold was set to 0.5.

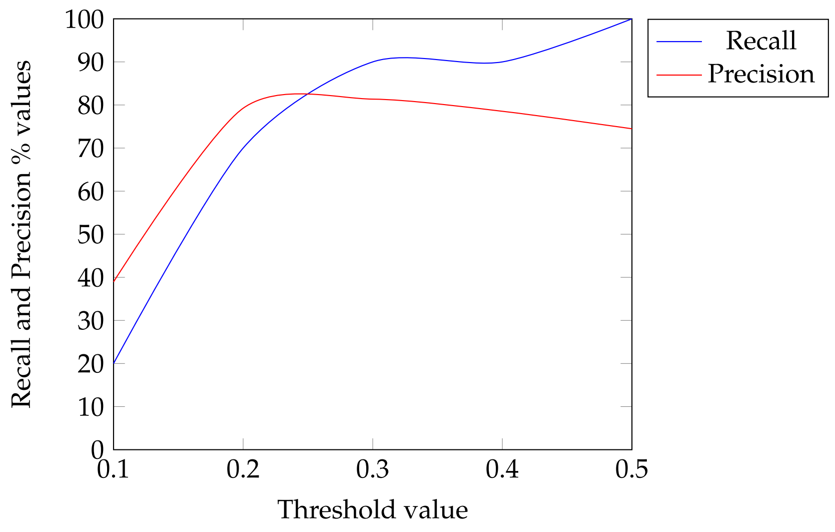


Figure 8. Average recall and precision values for a given threshold, when using PCA model M7.

Given that the detection algorithm resulted in high F1-scores when using both PCA models, we concluded that, overall, PCA worked well with the features selected to detect anomalies in JIT-MF logs. Crucially, by using PCA (an existing anomaly detection algorithm available to SOC analysts) with a 0.5 threshold, and using the set of features described in Table 5 and correlation settings defined in Section 5.3.2, the detection algorithm used by VEDRANDO could fully reconstruct the attack steps of benign app hijack attacks with a relatively high precision across all case studies based on evidence collected from JIT-MF logs.

We also evaluated the sensitivity of our detection and correlation algorithm to the threshold value given as a parameter using a Wilcoxon signed rank test (https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test). The results showed that the change-in-value of the F1-scores between the threshold values for K-Means, PCA, and Autoencoder were statistically significant. Therefore, when using such models, the algorithm is considered sensitive to the threshold set. The set of ARIMA_PLUS values was smaller; therefore, Wilcoxon values could not be calculated for this model. However, upon inspection, the F1-scores remained the same for the two most lenient threshold values.

6. Discussion

6.1. Deployment Feasibility

By enhancing an existing EDR, we showed that critical evidence of a benign app hijack found only in memory can be collected using tools already in use on Android phones, thus demonstrating the usability of our proposed solution in an operational setting. App-level virtualization is a key enabler of the feasibility of our proposed JIT-MF-enhanced EDR in this solution. It removes the need for app repackaging, which lends itself to a minimally invasive solution. While we demonstrated that this virtualization setup was successful on

84.8% of popular Android apps tested, the current app-level virtualization frameworks are still in their infancy. This means they are limited in functionality, rendering them non-functional or limited when using specific apps that require Google Play services, for instance. Further app-level virtualization limitations exist when the targeted apps are system apps, which this work has not addressed. Therefore, further development effort is needed to create more stable, open-source virtualization frameworks, before our proposed solution can be fully realized. Malware leverages attack vectors typically found on stock Android devices and apps, to enable inter-app communication with sensitive apps, which the attacker can leverage to carry out attack steps.

6.2. Privacy

Evidence collected by a JIT-MF-enhanced EDR may raise potential privacy issues. However, we argue that (i) our proposed solution aims to aid the response to end-users subjected to stealthy benign app hijack attacks, and therefore it is within their interest that the evidence is collected and analyzed at the time, and within the parameters, of the incident; (ii) privacy-aware forensics solutions exist [59] through which our proposed JIT-MF-enhanced EDR can collect the necessary evidence to reconstruct stealth attack steps, while still protecting sensitive information, to protect users' privacy; and (iii) the use of work profiles is recommended for enterprise settings (https://www.android.com/enterprise/work-profile/), which respect employee privacy through separate, dedicated work and personal profiles that give SOCs more flexibility with regards to privacy when retrieving data related to work profiles.

6.3. Anomaly Detection in JIT-MF Logs

In an enterprise setting, incident response and SOC teams use generic online services with machine learning capabilities that enable default machine learning techniques, which ease log analysis by providing anomaly detection for given data sets of forensic sources [45,60–62]. Similarly, in this work, we took a generic approach when selecting machine learning anomaly detection models and correlation techniques in our proposed detection algorithm, to show how JIT-MF logs enable the detection of anomalies that would otherwise remain hidden. The results showed promise and demonstrated that full attack step reconstruction is possible for specific parameters. Even when specific steps are missed during anomaly detection and correlation, critical evidence related to benign app hijacks in JIT-MF logs is still available for investigators to find. Therefore, further work in this area could focus on larger-scale studies looking into feature selection and identifying which machine learning models and parameters are appropriate for specific apps and attack scenarios.

7. Related Work

Table 10 compares related works from different categories that are closest to our work based on features (first column) derived from the set of requirements for VEDRANDO (**R1–R4**). Features include whether or not the related work: (i) targets Android OS (Android-inclusive); (ii) addresses acquisition from memory (memory acquisition); (iii) addresses benign app hijack threats, (iv) addresses the ephemerality of evidence in memory by triggering timely memory dumps (triggered dump); (v) works across multiple apps and attack scenarios (extensible); (vi) takes a minimally invasive approach regarding devices (stock devices), (vii) takes a minimally invasive approach regarding apps (stock apps); and (viii) performs anomaly detection (anomaly detection).

Table 10. Table showing a comparison of related works, based on features that are novel to this work.

Feature	**VEDRANDO**	**Forensic Logging**	**Memory Forensics**	**JIT-MF**	**Anomaly Detection**	**Log Reduction**	**Attack Investigation**
		[63–65]	[66–68]	[21,23–25]	[69–71]	[72–78]	[56,79]
Android inclusive	✓	✓	✓	✓	✕	✕	✓*
Memory acquisition (**R1**)	✓	✕	✓	✓	✕	✕	✕
Benign app hijack threat	✓	✕	✕	✓	✕	✕	✕
Triggered dump (**R1**)	✓	✕	✓*	✓	✕	✕	✓*
Extensible (**R2**)	✓	✓	✓	✓	✕	✕	✕
Stock devices (**R3**)	✓	✓	✕	✓	✕	✕	✕
Stock apps (**R3**)	✓	✓	✓	✕	✕	✕	✕
Anomaly Detection (**R4**)	✓	✕	✕	✕	✓	✕	✓

✓ shows that works in this category have these features. ✕ shows that works in this category *do not* have these features. ∗ shows that *some* works in this category have these features.

7.1. EDR and Memory Forensics

The use of memory forensics as a source of evidence for EDRs is not novel. Many related works and enterprise tools [63–68] leverage evidence in memory to help reconstruct attack steps, especially in the case of stealth attacks. Therefore, using memory forensics in Android devices to help reconstruct stealth attack steps is unsurprising. However, in this work, we have proposed a minimally invasive solution that works on stock Android devices and apps and that aims to collect evidence from memory in a timely manner, to address stealth attacks that aim to hide their attack steps.

Our work builds on previous research that introduced the JIT-MF framework [21,23–25], and demonstrated its ability to log evidence of a benign app hijack attack located in memory and that is not collected from any other forensic source. Previous work evaluated the impact of different trigger point types on the object (evidence) dumped and showed that JIT-MF logs contained critical evidence of app hijack attacks not collected by other state-of-the-art forensic sources. However, the limitations related to app repacking were not addressed, which meant that JIT-MF drivers had to circumvent any anti-repackaging techniques employed by the app. Thus far, the impact of events collected from JIT-MF logs has not been shown. The experimental results in this paper showed that collecting evidence of benign app hijack attacks from memory is possible through JIT-MF and is made feasible through a JIT-MF-enhanced EDR using infrastructure-based JIT-MF drivers. Furthermore, we showed that the logs produced by such an EDR allow for detecting anomalous behavior not found in other forensic sources and that this is the foundation for building the complete attack steps.

7.2. Anomaly Detection for Security

We note that the use of anomaly detection to detect malware has been presented in various works. Such works focused on detecting anomalous behavior in logs [69–71] and tackle problems related to feature selection, automatic parsing of different, high-volume logs, and finding appropriate anomaly detection methods. However, the premise of such work is that the log events on which anomaly detection models operate include events that indicate abnormal usage. In the case of stealthy Android attacks that hijack benign app functionality, we show that the logs of forensic sources typically collected do not include the necessary evidence to enable finding anomalous behavior and that timely collected evidence from memory could add the necessary context.

7.3. Log Reduction

As a result of aiming to collect additional evidence that discloses stealth attacks, our solution amasses more logs, which runs the risk of exceeding the storage capacity, thus reducing the feasibility and possibly amounting to a needle-in-a-haystack problem. A line of work exists focusing on reducing the log size of events, while preserving only a necessary smaller set to enable provenance tracking [72–78]. Previous use of JIT-MF introduced trigger point sampling, aiming to reduce app crashes, resulting in reduced load burden on the device as an added benefit. Thus, our proposed solution is orthogonal to these approaches and could incorporate them to reduce the storage of events.

7.4. Attack Investigation

UIScope [56] is a similar work that aims to retrieve additional context through evidence gathered from the UI, to perform a causality analysis of attacks that require user input on Windows computers. Similarly, it leverages and correlates UI events with other events (in this case, system events) to perform a causality analysis with accuracy and visibility. Our proposed solution also aims to gather additional context to reconstruct attack steps. It focuses on showing the feasibility and value of our approach on Android devices, which present more challenges in terms of feasibility, due to the restricted environment (unrooted devices). Difuzer [79] targets a similar problem to the one discussed in this paper, as it aims to detect a specific class of logic bombs (Suspicious Hidden Sensitive Operations—

SHSO) used in Android malware. Difuzer focuses on Android devices and aims to disclose suspicious operations that could indicate the possibility of an SHSO through anomaly detection. Similarly, our work focused on collecting evidence indicating the possibility of benign app hijacking. While Difuzer's primary goal is to detect SHSO, our proposed solution aims to collect the necessary evidence to make it possible for anomaly detection techniques and correlation algorithms to detect malicious behavior in the general case of a benign app hijack.

8. Conclusions

In this paper, we proposed VEDRANDO, an enhanced EDR for Android that comprises the timely collection of volatile memory artefacts and the detection of a class of stealth attacks that hijack benign Android applications. VEDRANDO highlights the critical role of evidence from memory in detecting and reconstructing the attack steps of stealth app hijack attacks that are not collected by the current state-of-the-art tools. By leveraging experimental techniques for timely memory collection and app-level virtualization, VEDRANDO can collect evidence from memory without requiring app repackaging or device rooting, thus ensuring the feasibility of our solution. VEDRANDO also uses existing anomaly detection methods and correlation techniques, typically available to SOC teams and investigators, to detect evidence and reconstruct the attack steps of stealthy benign hijack attacks. Our results show that deploying VEDRANDO is feasible, as it incurs minimal performance overheads, and JIT-MF driver development efforts can be eased through infrastructure-based JIT-MF drivers. Furthermore, our evaluation showed that, given a set of anomaly detection methods and parameters, VEDRANDO can effectively and precisely reconstruct attack steps up to the malware entry point for the class of stealth attacks that hijack benign messaging app functionality.

Author Contributions: Conceptualization, J.B., M.V., C.C., E.L, M.C; methodology, J.B., M.V., C.C., E.L., M.C.; investigation, J.B.; resources, J.B., M.V., C.C., E.L., M.C.; data curation, J.B.; writing—original draft preparation, J.B.; writing—review and editing, J.B., M.V., C.C., E.L., M.C.; visualization, J.B.; supervision, M.V., C.C., E.L., M.C.; project administration, M.V., M.C.; funding acquisition, M.V., C.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the DETECTIF (Digital Evidence Targeting covErt Cyberattacks through Timely Information Forensics) project. DETECTIF is funded by the Malta Council for Science and Technology (MCST) under the FUSION R&I: Research Excellence Programme, Grant Agreement No. REP-2022-007.

Data Availability Statement: Experiment artefacts can be found in the accompanying paper repository https://gitlab.com/bellj/vedrando/-/tree/main/forensic_artefacts_collected.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kotzias, P.; Caballero, J.; Bilge, L. How did that get in my phone? Unwanted app distribution on android devices. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 53–69.
2. Collier, N. Malware on the Google Play Store Leads to Harmful Phishing Sites. 2022. Available online: https://www.malwarebytes.com/blog/news/2022/11/malware-on-the-google-play-store-leads-to-harmful-phishing-sites (accessed on 18 February 2023).
3. The Mobile Malware Threat Landscape in 2022. 2022. Available online: https://securelist.com/mobile-threat-report-2022/108844/ (accessed on 28 March 2023).
4. BrasDex: A New Brazilian ATS Android Banker with Ties to Desktop Malware. 2022. Available online: https://www.threatfabric.com/blogs/brasdex-a-new-brazilian-ats-malware.html (accessed on 3 March 2023).
5. Look out for Octo's Tentacles! A New On-Device Fraud Android Banking Trojan with a Rich Legacy. 2022. Available online: https://threatfabric.com/blogs/octo-new-odf-banking-trojan.html (accessed on 3 March 2023).
6. Stefanko, L. Insidious Android Malware Gives up All Malicious Features But One to Gain Stealth. 2020. Available online: https://www.welivesecurity.com/2020/05/22/insidious-android-malware-gives-up-all-malicious-features-but-one-gain-stealth/ (accessed on 24 March 2021).

7. ThreatFabric. 2020-Year of the RAT. 2020. Available online: https://www.threatfabric.com/blogs/2020_year_of_the_rat.html (accessed on 24 March 2021).
8. FluBot Malware–All You Need to Know & to Act Now. 2021. Available online: https://www.threatmark.com/flubot-banking-malware/ (accessed on 6 March 2023).
9. Eventbot: A New Mobile Banking Trojan Is Born. 2020. Available online: https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born (accessed on 24 March 2023).
10. Fratantonio, Y.; Qian, C.; Chung, S.P.; Lee, W. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In Proceedings of the IEEE S&P, San Jose, CA, USA, 22–24 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1041–1057.
11. Li, T.; Wang, X.; Zha, M.; Chen, K.; Wang, X.; Xing, L.; Bai, X.; Zhang, N.; Han, X. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 829–844.
12. Yang, Y.; Zhang, Y.; Lin, Z. Cross Miniapp Request Forgery: Root Causes, Attacks, and Vulnerability Detection. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 3079–3092.
13. Triada: Organized Crime on Android. 2016. Available online: https://www.kaspersky.com/blog/triada-trojan/11481/ (accessed on 30 March 2023).
14. Shi, L.; Fu, J.; Guo, Z.; Ming, J. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In Proceedings of the MobiSys, Seoul, Republic of Korea, 17–21 June 2019; ACM: New York, NY, USA, 2019; pp. 222–235.
15. Diamantaris, M.; Papadopoulos, E.P.; Markatos, E.P.; Ioannidis, S.; Polakis, J. Reaper: Real-time app analysis for augmenting the Android permission system. In Proceedings of the ACM CODASPY, Dallas, TX, USA, 25–27 March 2019; pp. 37–48.
16. Wang, X.; Shi, S.; Chen, Y.; Lau, W.C. PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In Proceedings of the 29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, CA, USA, 24–28 April 2022; The Internet Society: Reston, VA, USA, 2022.
17. Leguesse, Y.; Vella, M.; Colombo, C.; Hernandez-Castro, J. Reducing the Forensic Footprint with Android Accessibility Attacks. In Proceedings of the STM, Guildford, UK, 17–18 September 2020; pp. 22–38.
18. Vella, M.; Rudramurthy, V. Volatile memory-centric investigation of SMS-hijacked phones: A Pushbullet case study. In Proceedings of the FedCSIS, Poznań, Poland, 9–12 September 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 607–616.
19. Case, A.; Richard, G.G., III. Memory forensics: The path forward. *Digit. Investig.* **2017**, *20*, 23–33. [CrossRef]
20. Case, A.; Maggio, R.D.; Firoz-Ul-Amin, M.; Jalalzai, M.M.; Ali-Gombe, A.; Sun, M.; Richard, G.G., III. Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics. *Comput. Secur.* **2020**, *96*, 101872. [CrossRef]
21. Bellizzi, J.; Vella, M.; Colombo, C.; Hernandez-Castro, J. Responding to Targeted Stealthy Attacks on Android Using Timely-Captured Memory Dumps. *IEEE Access* **2022**, *10*, 35172–35218. [CrossRef]
22. Aarness, A. What is EDR? Endpoint Detection & Response Defined. Available online: https://www.crowdstrike.com/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/ (accessed on 2 March 2023).
23. Bellizzi, J.; Vella, M.; Colombo, C.; Hernandez-Castro, J. Real-time triggering of Android memory dumps for stealthy attack investigation. In Proceedings of the NordSec, Virtual Event, 23–24 November 2020; pp. 20–36.
24. Bellizzi, J.; Vella, M.; Colombo, C.; Hernandez-Castro, J. Responding to Living-Off-the-Land Tactics using Just-In-Time Memory Forensics (JIT-MF) for Android. In Proceedings of the SECRYPT, Lieusant, Paris, 6–8 July 2021; pp. 356–369.
25. Bellizzi, J.; Vella, M.; Colombo, C.; Hernandez-Castro, J. Using infrastructure-based agents to enhance forensic logging of third-party applications. In Proceedings of the ICISSP, Lisbon, Portugal, 22–24 February 2023.
26. AppBrain: Monetize, Advertise and Analyze Android Apps. 2022. Available online: https://www.appbrain.com/stats/libraries (accessed on 23 August 2022).
27. Zhang, R.; Xie, M.; Bian, J. ReLF: Scalable Remote Live Forensics for Android. In Proceedings of the 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Shenyang, China, 20–22 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 822–831.
28. Falaki, H.; Mahajan, R.; Estrin, D. SystemSens: A tool for monitoring usage in smartphone research deployments. In Proceedings of the Sixth International Workshop on MobiArch, Bethesda, MD, USA, 28 June 2011; pp. 25–30.
29. Grover, J. Android forensics: Automated data collection and reporting from a mobile device. *Digit. Investig.* **2013**, *10*, S12–S20. [CrossRef]
30. Wagner, D.T.; Rice, A.; Beresford, A.R. Device analyzer: Understanding smartphone usage. In Proceedings of the Mobile and Ubiquitous Systems: Computing, Networking, and Services: 10th International Conference, MOBIQUITOUS 2013, Tokyo, Japan, 2–4 December 2013; Revised Selected Papers 10; Springer: Berlin/Heidelberg, Germany, 2014; pp. 195–208.
31. Hoog, A.; Cannon, T.; Anderson, T. AFLogical OSE: Open Source Android Forensics App and Framework. 2015. Available online: https://github.com/nowsecure/android-forensics (accessed on 18 February 2023).
32. Spolaor, R.; Dal Santo, E.; Conti, M. Delta: Data extraction and logging tool for android. *IEEE Trans. Mob. Comput.* **2017**, *17*, 1289–1302. [CrossRef]
33. Sazonov, D. Andriller-Android Forensic Tools. Available online: https://github.com/den4uklandriller (accessed on 18 February 2023).

34. Hargreaves, C.; Patterson, J. An automated timeline reconstruction approach for digital forensic investigations. *Digit. Investig.* **2012**, *9*, S69–S79. [CrossRef]
35. Chabot, Y.; Bertaux, A.; Nicolle, C.; Kechadi, M.T. A complete formalized knowledge representation model for advanced digital forensics timeline analysis. *Digit. Investig.* **2014**, *11*, S95–S105. [CrossRef]
36. Chabot, Y.; Bertaux, A.; Nicolle, C.; Kechadi, T. Automatic Timeline Construction and Analysis for Computer Forensics Purposes. In Proceedings of the 2014 IEEE Joint Intelligence and Security Informatics Conference, The Hague, The Netherlands, 24–26 September 2014; pp. 276–279. [CrossRef]
37. Du, X.; Hargreaves, C.; Sheppard, J.; Anda, F.; Sayakkara, A.; Le-Khac, N.A.; Scanlon, M. SoK: Exploring the state of the art and the future potential of artificial intelligence in digital forensic investigation. In Proceedings of the 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, 25–28 August 2020; pp. 1–10.
38. Studiawan, H.; Sohel, F.; Payne, C. A survey on forensic investigation of operating system logs. *Digit. Investig.* **2019**, *29*, 1–20. [CrossRef]
39. Team, D. DroidPlugin. 2020. Available online: https://github.com/DroidPluginTeam/DroidPlugin (accessed on 3 December 2021).
40. Jining Luohe Network Technology Co., Ltd. 2020. VirtualApp. 2021. Available online: https://github.com/asLody/VirtualApp (accessed on 3 December 2021).
41. Shi, L.; Ming, J.; Fu, J.; Peng, G.; Xu, D.; Gao, K.; Pan, X. Vahunt: Warding off new repackaged android malware in app-virtualization's clothing. In Proceedings of the ACM SIGSAC, Virtual Event, 9–13 November 2020; pp. 535–549.
42. Chandola, V.; Banerjee, A.; Kumar, V. Anomaly detection: A survey. *Acm Comput. Surv. (Csur)* **2009**, *41*, 1–58.
43. Catillo, M.; Pecchia, A.; Villano, U. AutoLog: Anomaly detection by deep autoencoding of system logs. *Expert Syst. Appl.* **2022**, *191*, 116263.
44. Meng, W.; Liu, Y.; Zhu, Y.; Zhang, S.; Pei, D.; Liu, Y.; Chen, Y.; Zhang, R.; Tao, S.; Sun, P.; et al. LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In Proceedings of the IJCAI, Macao, China, 10–16 August 2019; Volume 19, pp. 4739–4745.
45. Splunk-Machine Learning. Available online: https://docs.splunk.com/Documentation/SplunkCloud/9.0.2303/Search/MachineLearning (accessed on 20 February 2023).
46. Gustuff: Weapon of Mass Infection. 2019. Available online: https://www.group-ib.com/blog/gustuff (accessed on 20 February 2023).
47. BlackRock-The Trojan That Wanted to Get Them All. 2020. Available online: https://www.threatfabric.com/blogs/blackrock_the_trojan_that_wanted_to_get_them_all.html (accessed on 20 February 2023).
48. Campbell, C.; Graeber, M. Living Off the Land: A Minimalist's Guide to Windows Post-Exploitation. 2013. Available online: http://www.irongeek.com (accessed on 24 March 2021).
49. Mohamed, M.; Shrestha, B.; Saxena, N. Smashed: Sniffing and manipulating android sensor data for offensive purposes. *IEEE Trans. Inf. Forensics Secur.* **2016**, *12*, 901–913. [CrossRef]
50. Maruyama, S.; Wakabayashi, S.; Mori, T. Tap'n ghost: A compilation of novel attack techniques against smartphone touchscreens. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 620–637.
51. Cohen, M.I.; Bilby, D.; Caronni, G. Distributed forensics and incident response in the enterprise. *Digit. Investig.* **2011**, *8*, S101–S110.
52. Alecci, M.; Cestaro, R.; Conti, M.; Kanishka, K.; Losiouk, E. Mascara: A Novel Attack Leveraging Android Virtualization. *arXiv* **2020**, arXiv:2010.10639.
53. Team, P.T. 2022 Recap-Mobile Malware Threat Landscape. Available online: https://blog.polyswarm.io/2022-recap-mobile-malware-threat-landscape (accessed on 28 March 2023).
54. Pei, K.; Gu, Z.; Saltaformaggio, B.; Ma, S.; Wang, F.; Zhang, Z.; Si, L.; Zhang, X.; Xu, D. Hercule: Attack story reconstruction via community discovery on correlated log graph. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, 5–8 December 2016; pp. 583–595.
55. Alsaheel, A.; Nan, Y.; Ma, S.; Yu, L.; Walkup, G.; Celik, Z.B.; Zhang, X.; Xu, D. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In Proceedings of the USENIX Security Symposium, Virtual Event, 11–13 August 2021; pp. 3005–3022.
56. Yang, R.; Ma, S.; Xu, H.; Zhang, X.; Chen, Y. UIScope: Accurate, Instrumentation-free, and Visible Attack Investigation for GUI Applications. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
57. Ruggia, A.; Possemato, A.; Merlo, A.; Nisi, D.; Aonzo, S. Android, notify me when it is time to go phishing. In Proceedings of the EUROS&P 2023, 8th IEEE European Symposium on Security and Privacy, Delft, The Netherlands, 3–7 July 2023; IEEE: Piscataway, NJ, USA, 2023.
58. What's New with BigQuery ML: Unsupervised Anomaly Detection for Time Series and Non-Time Series Data. 2021. Available online: https://cloud.google.com/blog/products/data-analytics/bigquery-ml-unsupervised-anomaly-detection (accessed on 15 April 2023).
59. Englbrecht, L.; Pernul, G. A privacy-aware digital forensics investigation in enterprises. In Proceedings of the 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, 25–28 August 2020; pp. 1–10.
60. Mandiant-Digital Threat Monitoring. Available online: https://www.mandiant.com/advantage/digital-threat-monitoring (accessed on 20 February 2023).

61. Elastic-Observability Module. Available online: https://www.elastic.co/guide/en/kibana/current/xpack-ml-anomalies.html(accessed on 20 February 2023).
62. Sumo Logic-Machine Learning Analytics. Available online: https://www.sumologic.com/solutions/machine-learning-powered-analytics/ (accessed on 20 February 2023).
63. Team, R. Rekall memory forensic framework: About the rekall memory forensic framework. *Retrieved March* **2015**, *13*, 2015.
64. GRR Rapid Response Framework. Available online: https://grr-doc.readthedocs.io/ (accessed on 15 February 2023).
65. Velociraptor-Document and Souce Code. Available online: https://github.com/Velocidex/velociraptor (accessed on 15 February 2023).
66. Yuan, X.; Setayeshfar, O.; Yan, H.; Panage, P.; Wei, X.; Lee, K.H. Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging. In Proceedings of the ACM ASIACCS, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 666–677.
67. Ali-Gombe, A.; Sudhakaran, S.; Case, A.; Richard, G.G., III. DroidScraper: A tool for Android in-memory object recovery and reconstruction. In Proceedings of the RAID, Beijing, China, 23–25 September 2019; pp. 547–559.
68. Bhatia, R.; Saltaformaggio, B.; Yang, S.J.; Ali-Gombe, A.I.; Zhang, X.; Xu, D.; Richard, G.G., III. Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
69. Kuppa, A.; Grzonkowski, S.; Asghar, M.R.; Le-Khac, N.A. Finding Rats in Cats: Detecting Stealthy Attacks using Group Anomaly Detection. *arXiv* **2019**, arXiv:1905.07273.
70. Liu, Z.; Qin, T.; Guan, X.; Jiang, H.; Wang, C. An Integrated Method for Anomaly Detection From Massive System Logs. *IEEE Access* **2018**, *6*, 30602–30611. [CrossRef]
71. Guo, H.; Yuan, S.; Wu, X. Logbert: Log anomaly detection via bert. In Proceedings of the 2021 international joint conference on neural networks (IJCNN), Shenzhen, China, 18–22 July 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–8.
72. Hassan, W.U.; Bates, A.; Marino, D. Tactical provenance analysis for endpoint detection and response systems. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1172–1189.
73. Ma, S.; Zhai, J.; Kwon, Y.; Lee, K.H.; Zhang, X.; Ciocarlie, G.; Gehani, A.; Yegneswaran, V.; Xu, D.; Jha, S. Kernel-supported cost-effective audit logging for causality tracking. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, USA, 11–13 July 2018; pp. 241–254.
74. Lee, K.H.; Zhang, X.; Xu, D. LogGC: Garbage collecting audit log. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 1005–1016.
75. Xu, Z.; Wu, Z.; Li, Z.; Jee, K.; Rhee, J.; Xiao, X.; Xu, F.; Wang, H.; Jiang, G. High fidelity data reduction for big data security dependency analyses. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 504–516.
76. Muniswamy-Reddy, K.K.; Holland, D.A.; Braun, U.; Seltzer, M.I. Provenance-aware storage systems. In Proceedings of the Usenix Annual Technical Conference, General Track, Boston, MA, USA, 30 May–3 June 2006; pp. 43–56.
77. Hassan, W.U.; Aguse, L.; Aguse, N.; Bates, A.; Moyer, T. Towards scalable cluster auditing through grammatical inference over provenance graphs. In Proceedings of the Network and Distributed Systems Security Symposium, San Diego, CA, USA, 18–21 February 2018.
78. Hossain, M.N.; Wang, J.; Sekar, R.; Stoller, S.D. Dependence-preserving data compaction for scalable forensic analysis. In Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 1723–1740.
79. Samhi, J.; Li, L.; Bissyandé, T.F.; Klein, J. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 723–735.