Using Infrastructure-Based Agents to Enhance Forensic Logging of Third-Party Applications

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

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

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

Keywords: Enhanced Forensic Logging, Deeper Endpoint Visibility, Digital Forensics, Android Security.

Abstract: Logs are the primary data source forensic analysts use to diagnose and investigate attacks on deployed applications. Since the default logs may not include all application events required during an investigation, application-specific forensic logging agents are used to forensically enhance third-party applications post-deployment and ensure that any critical events are logged. However, developing such application-specific agents is impractical as this relies on application-specific knowledge requiring significant code comprehension efforts. Furthermore, the resulting forensic logging agents are likely to break compatibility between application versions and across applications; thus, requiring the time-consuming process of agent re-development much more frequently. We propose a more practical approach to developing forensic logging agents that leverages commonly-used underlying infrastructure, which is more stable across application versions and common across different applications. We evaluate our approach in the context of enhanced logging of Android messaging apps. Our results show that this approach can be used to develop logging agents that work across multiple apps while preserving the accuracy of the logs generated, thus mitigating the challenges associated with forensically enhancing third-party applications.

1 INTRODUCTION

Incident responders use logs to investigate attacks on deployed applications. Investigators rely on application logs to understand both the entry point of an attack on an application and its ramifications; that is, what damages have been inflicted and which data assets are being targeted (Ma et al., 2015). However, it is difficult to know a priori where logs are needed to ensure attack-related events are logged in the future. Exhaustively recording all possible application events is infeasible due to the resulting overheads. As a result of these issues, applications can produce lots of log entries, but rarely the relevant ones needed to investigate an attack. Furthermore, existing logs may be inaccessible, for instance, in a mobile investigation involving an unrooted device.

Several tools, including GRR Rapid Response[1] (Cohen et al., 2011) and Velociraptor IR[2] use client device agents for end-point visibility purposes. These agents are installed within third-party applications or on devices as part of the forensic readiness stage to aid in Incident Response. They collect files, application and system logs, and outputs from live forensics commands that harbour indicators of compromise to feed them back to a central investigation node. Making use of additional agents that are inserted inside third-party application processes (for which access to the source code is restricted) carries the potential to enhance logging with deeper visibility. While opting for application-specific agents may seem the most obvious solution to log application-related events, it can be impractical. Such agents rely on application-specific objects, resources and logic that may change when newer versions are released. Therefore, not only do application-specific agents require significant code comprehension effort for every application, their reliance on application-specific logic means that application-specific logging agents will likely break compatibility across applications and versions.

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

[2]https://github.com/Velocidex/velociraptor

Bellizzi, J., Vella, M., Colombo, C. and Hernandez-Castro, J.
Using Infrastructure-Based Agents to Enhance Forensic Logging of Third-Party Applications.
DOI: 10.5220/0011634700003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 389-401
ISBN: 978-989-758-624-8; ISSN: 2184-4356

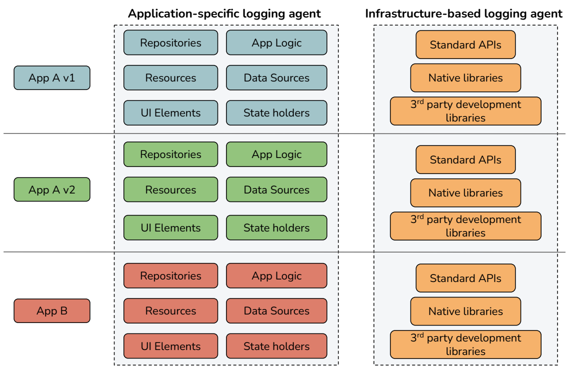

Figure 1: Forensic logging agents leveraging application-specific codebase versus agents leveraging common underlying infrastructure-interfacing code that is expected to be more stable across versions (App A v1, v2) and applications (App A, B). Different colour codes signify application-specific codebases, whereas the same-coloured codebases reflect common APIs across applications and versions.

This paper proposes a new approach to address challenges that arise when relying on application-specific logic to build post-deployment forensic logging agents for third-party applications, for which access to the source code is limited. Figure 1 presents the main insight underpinning our proposed solution, which aims to leverage common, widespread infrastructure across different application versions and applications altogether. While application-specific components of an application may change when newer versions are released (see App A v1 and v2 in Figure 1) and differ across different applications (see App A and B in Figure 1), infrastructure-centric components such as native libraries and APIs, are more likely to remain compatible with other applications and versions. As a direct benefit of our approach, code comprehension effort is significantly reduced since i) common underlying infrastructure is publicly well-documented, ii) the focus is on a subset of the code that interacts with infrastructure code, and iii) the resulting agent has the potential to work across versions and other applications that use common underlying infrastructure. Furthermore, infrastructure logic is expected to be more stable and backward-compatible, whereas application internals may change between versions. Therefore, updates to infrastructure-based agents will not be required as frequently as application-specific agents that rely on application internals which are constantly in flux and out of the logging agent developer's control.

While this approach has several benefits, an essential requirement is that accuracy in logged events is not diminished. Since infrastructure-based logging agents focus on a reduced subset of the application codebase, the control over application-specific log entry metadata that needs to be logged will also be reduced, which may impact the accuracy of the log entries produced. With this in mind, we set out to evaluate our approach in the context of Android messaging applications, a common target for attacks (Flubot, 2021; Amer Owaida, 2021; Jon Russell, 2012), and make the following contributions:

- We verify the potential for infrastructure-centric logging agents to be compatible across different apps of the same category through AppBrain and determine the stability of underlying infrastructure among selected application versions by performing a static analysis of application versions spread across five years.
- We produce a prototype logging agent based on common infrastructure logic for Android messaging applications that can operate on three different applications and conduct a qualitative study to assess the accuracy of the log entries produced.

2 BACKGROUND

The use of log enhancement through application-specific agents is not just a forensic visibility problem. Logs are the primary data source engineers, developers, and analysts use to diagnose issues in deployed applications, such as performance issues, bugs or even evidence of attacks (Toslali et al., 2021; Yuan et al., 2012a; Yuan et al., 2012b; Zhao et al., 2017; Mace et al., 2015). However, it is difficult to know a priori where logs are needed to help diagnose problems that may occur in the future. Post-deployment log enhancement refers to logging third-party application events after the application, for which access to the source code is restricted, has been deployed so that relevant events related to unforeseen issues may be logged and analysed. This enhancement is typically carried out through application-specific agents that enhance application logging by relying on application-specific logic. However, relying on application-specific knowledge requires reverse engineering applications, which introduces several challenges, including i) comprehending compiled code, ii) bypassing any anti-tampering solutions present in the application code and finally, iii) complying with fair-use reverse engineering.

2.1 Log Collection for Forensic Analysis

Similar challenges apply to forensic log collection. Cyberattack incidents increasingly use malware that interacts with benign applications to hide their presence or escalate privileges, as well as carry out on-device fraud (ThreatLandscape, 2022; Octo, 2022). Log collection is crucial in a forensic investigation to gather all possible evidence to reconstruct the attack steps. Unfortunately, critical evidence is not always located in the collected logs, especially in cases involving stealthy malware that hides malicious behaviours to minimise suspicion and prolong its lifetime, thus requiring late detection. Advanced persistent threats (APTs) initiated by resourceful attackers and fileless malware operating in memory fall into this category. Other advanced stealth techniques have emerged through attack vectors that misuse the legitimate actions of targeted benign applications for financial or personal data theft.

In the case of malicious interactions with pre-installed, benign applications, entries in application logs are crucial as they contain evidence of application activities carried out seemingly by the benign application that could reconstruct the attack steps involving application misuse. Even if application logs are available, the investigators are still at the mercy of the application developers who control the contents of the log entries. Typically production application logs are used for diagnostic purposes, which may result in log entries that do not produce the evidence required in the case of application misuse. On the other hand, it is infeasible to introduce exhaustive logging of all possible application events by default, as this would incur significant overhead on all application users, including those unaffected by a stealthy attack.

In this case, investigators require logging agents installed inside third-party application processes. Yet, the lack of collaboration with application developers can complicate matters. Such agents would rely on application-specific logic to log related events. Relying on application-specific knowledge not only introduces the previously-mentioned challenges associated with reverse engineering applications but also introduces issues related to practicality.

2.2 JIT-MF: A Specific Case of Post-Deployment Forensic Log Enhancement on Android

Just-in-Time Memory Forensics (JIT-MF) (Bellizzi et al., 2020; Bellizzi. et al., 2021; Bellizzi et al., 2022) is a framework that enables live process memory forensics in a setting involving stealthy malware that delegates their attack steps to benign applications. JIT-MF was conceived to be adopted by incident response tools for stock smartphones without breaking any of their security controls. Rather, given that its primary purpose is to aid device owners in recovering from cyberattacks, 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 agents referred to as $JIT-MF$ *Drivers* that timely dump evidence from memory, using app instrumentation to avoid device rooting. These non-intrusive agents are installed within third-party applications to record additional data from memory, as part of the forensic readiness stage of incident response, *without changing the application code beyond hooking*. 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 signify that *Evidence_objects* are in memory, and hence *when* memory dumps should be triggered. Therefore, trigger points are crucial to *timely* dump evidence objects in memory.

JIT-MF Logs. Taking the case of a stealthy 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. To aid investigation efforts, 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 *trigger_point* can be defined as any operation handling the *MessageObject*. A sample of the resulting JIT-MF log entries is shown in Listing 1, whereby each entry consists of metadata derived from the evidence object (in this case *MessageObject*) as dumped from 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.

On the other hand, identifying infrastructure-centric evidence objects, for example, a *String* object, is expected to be simpler and far less likely to change when compared to an application-specific object. However, log entry completeness becomes an issue due to the level of information derivable from a generic *String* object.

Listing 1: JIT-MF log entry sample generated while using WhatsApp, Telegram and Signal Android apps (Bellizzi et al., 2022).

```
{"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"}}
```

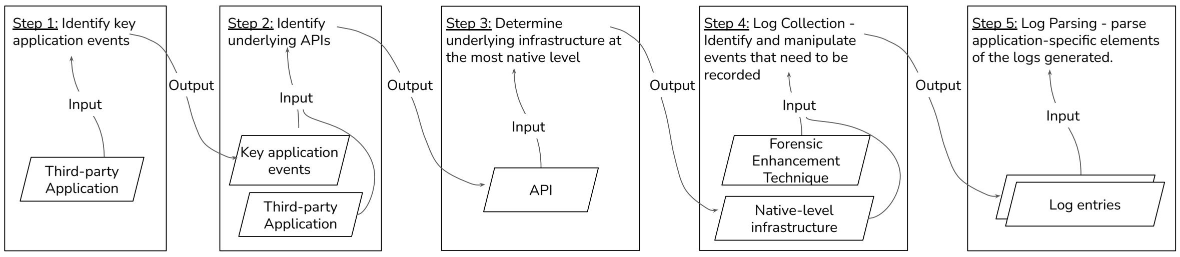
Step 1: Identify key application events
Input
Third-party Application
Output
Step 2: Identify underlying APIs
Input
Key application events
Third-party Application
Output
Step 3: Determine underlying infrastructure at the most native level
Input
API
Output
Step 4: Log Collection - Identify and manipulate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

Figure 2: Proposed process for creating and deploying infrastructure-based logging agents.

3 OVERVIEW

The main aim of this work is to present and evaluate a modified approach towards developing post-deployment third-party application logging agents that leverages the common subset of the applications' codebase that interacts with commonly-used infrastructure. As shown in Figure 1, the underlying infrastructure is generally more stable and widespread across different applications and versions, allowing infrastructure-based logging agents to remain usable across different applications and versions (unlike application-specific agents, which need to be developed anew). Figure 2 presents the process for creating *infrastructure-based forensic logging agents* as part of this new proposed approach.

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

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

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

Step 4: Given a forensic enhancement technique and the underlying infrastructure identified, publicly available documentation of the interface exposed by the infrastructure can be used to determine which infrastructure calls need to be logged to produce log entries containing the identified key application events. Since the focus is on infrastructure-based events, this step involves analysing only a subset of the application codebase that is typically publicly well-documented and can be *common* for applications that use the same underlying infrastructure.

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

4 EVALUATION

We evaluate our proposed approach of infrastructure-based forensic logging agents in the context of Android messaging apps and use JIT-MF, a forensic enhancement technique for third-party Android applications whose effectiveness has been demonstrated in previous works (Bellizzi et al., 2020; Bellizzi. et al., 2021; Bellizzi et al., 2022). We first conduct a preliminary analysis to select a suitable infrastructure candidate used to carry out core functionality in messaging apps. Then, we run experiments to answer the following questions: *Is common infrastructure usage prevalent across different versions of a messaging app?* (Q1) *Can infrastructure-based agents work across different Android messaging apps while maintaining the same accuracy as application-specific agents?* (Q2)

4.1 Common Infrastructure Adoption

Our preliminary analysis aims to identify which infrastructure is commonly-used among messaging apps to ensure that an infrastructure-based agent can function across app versions and on multiple apps. Not only does this infrastructure need to be in use by the app, but it also needs to be taking an *active role* when critical events that need to be logged are taking place. We consider the data provided by AppBrain (AppBrain, 2022), a service that provides statistics on the Android application's ecosystem including library adoption by different apps in different categories. In this case, we focus specifically on Android messaging apps, per the scope of this evaluation.

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

Results. At the time of our analysis, AppBrain statistics showed that the most widely used network library was Retrofit[3]. In contrast, the most widely used database infrastructure was Android Architecture Components[4], which at its most native level (see Step 3 Figure 2) refers to storage management through an SQLite Database[5]. AppBrain statistics revealed that at the time, 86.62% of communication (messaging) apps used Android Architecture Components, while only 14.6% used Retrofit.

Threats to Validity. AppBrain is closed-source; however, it provides insight into how it derives information regarding Android apps on Google Play-Store. It analyses all Android apps on Google Play and performs some analysis on the package file of the app (APK). Statistics regarding development tools and libraries usage are obtained by matching package names inside the apps "with known package names from development tools. Therefore, these statistics reflect whether the code of a certain library is present in an app." (AppBrain, 2022). While this does not guarantee that a particular application actively uses a library, "it still gives a good idea of the market share and what is the list of the top Android development tools." (AppBrain, 2022).

4.2 Prevalence Across App Versions

Through the results shown in the previous section we determine that storage libraries, specifically managed through SQLite, are the most commonly-used storage library and surpass the usage of network libraries in

[3]https://square.github.io/retrofit/
[4]https://www.appbrain.com/stats/libraries
[5]https://developer.android.com/training/data-storage/sqlite

messaging apps by 68.02%. JIT-MF has been previously evaluated (Bellizzi et al., 2022) using app-specific drivers (logging agents) for WhatsApp, Telegram and Signal, and we therefore use statistics provided by AppBrain to assert that this underlying infrastructure is present in the ***latest*** version of each of these apps.

AppBrain does not provide statistics on the previous versions of Android applications. Therefore, we use a quantitative static analysis approach within a qualitative set of applications (comprising of WhatsApp, Telegram and Signal) to assess the longevity of an SQLite infrastructure-based forensic logging agent across application versions, thus answering Q1.

Methodology. We answer this question in two parts. First, we assess the prevalence of SQLite library usage across the previous Telegram, WhatsApp and Signal versions by statically checking for the presence of code that uses this library. Second, we compare the frequency of releases between the SQLite library and each of the apps to indicate the stability that infrastructure-based agents relying on public-facing infrastructure codebase can provide compared with application-specific logging agents relying on an application-specific codebase.

Previous versions of each app were obtained from August 2017 every six months up until August 2022 (10 past versions per app, 30 apps in total), using APKCombo[6], a repository for apps and their previous versions. Versions of the apps dating more than six months prior could not be installed successfully due to limitations and restrictions presented by each app. Telegram was the only exception which allowed versions from six-month prior to be run. Due to this limitation, we conduct a static check to assert SQLite library usage in the decompiled sources of an app's version. This check is carried out based on a signature that encapsulates how a library can be used in an Android app. To use a library, Android apps can either: i) Call Android's API wrapper or ii) Use a custom implementation that interfaces with the infrastructure using JNI by including a shared object in the APK.

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

[6]https://apkcombo.com/

Listing 2: Signature for checking the presence of SQLite usage in an Android app package (APK).

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

Results. Table 1 shows the results obtained when the signature in Listing 2 is used to check for SQLite usage within the decompiled application versions obtained. These results show that bindings with the common underlying SQLite infrastructure are present across all apps and previous versions from the last five years, even if the interfacing method has changed. Therefore, selecting an infrastructure-based logging agent based on SQLite in the case of WhatsApp, Telegram and Signal is likely to remain compatible with upcoming versions of the apps.

Table 2 shows the frequency by which newer versions of the apps and SQLite are released. Application-specific logging agents for WhatsApp, Telegram and Signal are based on application codebases that are updated on average every 6 - 15 days. A compatibility test would therefore be required accordingly to ensure that application-specific logic within the agent was not altered in the update. On the other hand, updates to the SQLite library are not only much less frequent but also optional and hence not necessarily reflected in the apps that use them. Furthermore, in the case of SQLite, stable interfaces are maintained indefinitely in a backward-compatible way[7], which means that the agent relying on an older version of SQLite will not remain compatible, even if the infrastructure is updated.

Threats to Validity. We rely on a static signature that follows logic on how apps interact with the underlying infrastructure. However, this does not guarantee that the developer is using the code, as there is also the possibility that the code or libraries found reflect dead or legacy code, which cannot be distinguished statically.

4.3 Android Messaging Case Study

We conduct a qualitative case study to evaluate whether or not infrastructure-based agents can work across applications while maintaining the same level of accuracy in the logs produced as application-specific agents, and thereby answering Q2. Therefore,

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

Table 1: The table shows whether or not the SQLite usage signature was matched in different versions of Signal, Telegram and WhatsApp, since 2017.

Release Date	App version	Found SQLite function calls in decompiled smali code (1)	Found shared object in library folder (2)
23-08-2017	Signal v.4.9.9	✓	✗
28-02-2018	Signal v.4.16.9	✓	✓
06-08-2018	Signal v.4.24.8	✓	✓
09-02-2019	Signal v.4.33.5	✓	✓
09-08-2019	Signal v.4.45.2	✓	✓
12-02-2020	Signal v.4.55.8	✓	✓
20-08-2020	Signal v.4.69.4	✓	✓
18-02-2021	Signal v.5.4.6	✓	✓
20-08-2021	Signal v.5.21.5	✓	✓
18-02-2022	Signal v.5.32.7	✓	✓
05-08-2017	Telegram v.4.2.2	✗	✓
19-02-2018	Telegram v.4.8.4	✗	✓
30-08-2018	Telegram v.4.9.1	✗	✓
09-02-2019	Telegram v.5.3.1	✗	✓
24-08-2019	Telegram v.5.10.0	✗	✓
16-02-2020	Telegram v.5.15.0	✗	✓
16-08-2020	Telegram v.7.0.0	✗	✓
18-02-2021	Telegram v.7.4.2	✗	✓
07-08-2021	Telegram v.7.9.3	✗	✓
14-02-2022	Telegram v.8.5.2	✗	✓
11-08-2017	WhatsApp v.2.17.296	✓	✓
09-02-2018	WhatsApp v.2.18.46	✓	✓
18-08-2018	WhatsApp v.2.18.248	✓	✓
08-02-2019	WhatsApp v.2.19.34	✓	✓
07-08-2019	WhatsApp v.2.19.216	✓	✓
13-02-2020	WhatsApp v.2.20.22	✓	✓
05-08-2020	WhatsApp v.2.20.196.16	✓	✗
06-02-2021	WhatsApp v.2.21.3.13	✓	✗
09-08-2021	WhatsApp v.2.21.17.1	✓	✗
17-02-2022	WhatsApp v.2.22.4.75	✓	✗

Table 2: The frequency of Signal, Telegram, WhatsApp and SQLite library version releases, since 2017.

Codebase	Average Release time (in days) over the last 5 years
WhatsApp	6.324
Telegram	14.917
Signal	7.319
SQLite	39.48[*]

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

the case study involved two Google Pixel 3XL emulators running Android 10 (API 29). Telegram v.8.8.5, Whatsapp v.2.22.17.70 and Signal v.5.44.4 were installed on both emulators. JIT-MF was used as a forensic enhancement tool that equips these apps on one of the emulators (Device A) with a logging agent (JIT-MF Driver).

Methodology. Existing JIT-MF drivers from previous research[8] that cater for messaging hijacks on Telegram, WhatsApp and Signal are used and a new SQLite infrastructure-based driver[9] is developed that can be used with all three apps. The applications are equipped with their respective application-specific JIT-MF driver and the same SQLite infrastructure-based driver. AndroidViewClient was used to simulate normal messaging traffic, whereby 20 messages were sent, and 20 messages were received by Device A. Each message was formulated as follows: *Normal_Message_ < #msgnumber >*.

The log entries produced by the drivers are expected to include any messages sent and received by the enhanced app. Therefore, to evaluate whether an

[8]https://gitlab.com/mobfor/jitmf_experiments_resources

[9]https://gitlab.com/bellj/infrastructure_based_agents

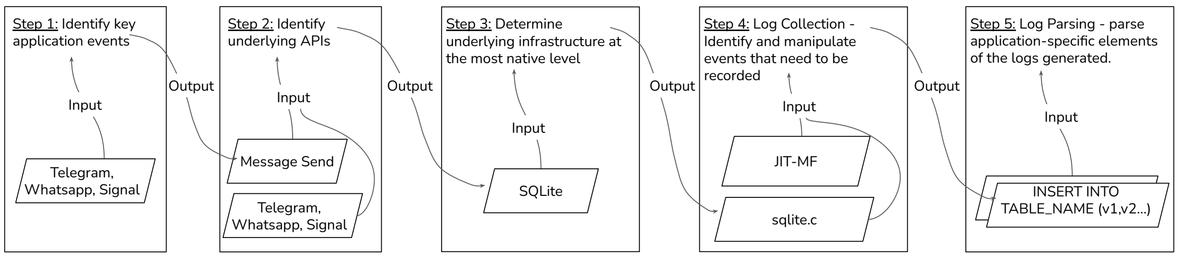
Step 1: Identify key application events
Input
Telegram, Whatsapp, Signal
Output
Step 2: Identify underlying APIs
Input
Message Send
Telegram, Whatsapp, Signal
Output
Step 3: Determine underlying infrastructure at the most native level
Input
SQLite
Output
Step 4: Log Collection - Identify and manipulate events that need to be recorded
Input
JIT-MF
sqlite.c
Output
Step 5: Log Parsing - parse application-specific elements of the logs generated.
Input
INSERT INTO TABLE_NAME (v1,v2...)

Figure 3: Process for creating and deploying SQLite infrastructure-based logging agents for Telegram, WhatsApp and Signal, using SQLite as infrastructure and JIT-MF as the forensic enhancement tool.

SQLite infrastructure-based driver can maintain the same level of accuracy in its log entries as application-specific drivers, we compare these log entries to those produced by application-specific drivers.

Logging Agent Development. JIT-MF drivers are the logging agents used in JIT-MF. We created a new SQLite infrastructure-based JIT-MF driver as an infrastructure-based logging agent aimed at being able to enhance all three apps. Following the steps shown in Figure 2, the SQLite infrastructure-based driver was developed, as shown in Figure 3. We identify message send events as the key application event that the SQLite infrastructure-based agent needs to log. Log entries produced through JIT-MF are invoked at trigger points containing evidence objects. In the case of an SQLite infrastructure-based JIT-MF driver, trigger points were defined as the functions exposed by the `sqlite` C interface[10] and evidence objects comprise of a parameter within these functions that contains the SQL statements executed.

Log Collection. The publicly available documentation for SQLite indicates that prepared statements allow applications to execute all SQL statements. Furthermore, the second parameter of any prepared statement contains the SQL query to be executed. Subsequent bind functions (`BIND_INT,BIND_TEXT,BIND_BLOB`) are used to populate the parameterised values of the query, depending on a query ID[11]. Therefore log entries produced by SQLite infrastructure-based logging agent should consist of SQL statements that were executed to populate messages in the messaging app local database.

Log Parsing. An offline component was required to parse the log entries produced. While the SQLite infrastructure-based JIT-MF driver was the same for all three apps, the log entries produced contained application-specific knowledge that required unique parsing logic per app. For instance, not all apps internals use the same SQL statements to populate their respective local databases. However, SQLite databases include a schema that describes the tables within a database. This means that the developed parser could use this information to carve out the necessary portions from the log entries accordingly.

Listing 3 shows three unparsed log entries generated by an SQLite infrastructure-based driver, while sending messages from WhatsApp, Telegram and Signal, respectively. Each entry comprises of an `INSERT | REPLACE` statement, signifying a new message that will be inserted in the local database. Application-specific parsers were developed which could: i) identify the names of the `message` table unique to each app (`message_ftsv2`, `messages_v2` and `sms`), and ii) map the appropriate *field values* with the key application event metadata; for instance, the second field value of a **`REPLACE INTO`** `messages_v2` table generated when using Telegram reflects the time at which a message was sent. The respective parsed log entries are shown in Listing 4.

Results. Publicly-available documentation allowed the development of an SQLite infrastructure-based logging agent without requiring application-specific code reversing. The same SQLite infrastructure-based JIT-MF driver was effective on all three apps; however, as expected, offline application-specific log parsing was required per app.

We compare the parsed output generated by the SQLite infrastructure-based logging agent with that produced by the application-based agents. The final column in Table 3 shows that the SQLite infrastructure-based agent was able to produce the same number of log entries containing the same text metadata, as each of the application-specific drivers. While *text* metadata in log entries generated by the SQLite infrastructure-based logging agent (Listing 4) and application-based agents (Listing 1) is the same, closer inspection of the two sample log entry outputs generated, shows some difference in the other key

[10]https://www.sqlite.org/c3ref/funclist.html
[11]https://www.sqlite.org/c3ref/stmt.html

Listing 3: Unparsed log entries generated by infrastructure-based agent for WhatsApp (1), Telegram (2) and Signal (3) respectively.

```
1  INSERT INTO message_ftsv2(fts_jid,docid,content,fts_namespace) VALUES (0 b,null,normal_message_1,0 b)
2  REPLACE INTO messages_v2 VALUES(2328, 1679923803, 2, 0, 1662483779, n8\"QY[!d\"QY[!dC}cNormal_message_1 , 0, 0,
         18446744073709552000, NULL, 0, 0, 0, undefined, 0, 0, 0, undefined)
3  INSERT INTO sms(thread_id,subscription_id,address,protocol,expires_in,server_guid,date_sent,body,date,read,type,
         unidentified,date_server,reply_path_present,service_center,address_device_id) VALUES (3,18446...0,3, 31337,0,d500 3
         b3d-ce8e-41cf-ba67-1cf592fa81c2,1662485982439,Normal_message_1Â§,1662485983160,0,10485780, 1,1662485974012,1,GCM,1)
```

Listing 4: JIT-MF log entry sample generated while using WhatsApp, Telegram and Signal, produced through an SQLite infrastructure-based logging agent.

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

Table 3: The table shows the maximum lines of code (LoC) that need to be analysed to develop an application-specific agent and an SQLite infrastructure-based agent, along with the percentage of log entries retrieved by an SQLite infrastructure-based agent, when compared with log entries generated using an application-specific agent.

Application	**Maximum LoC analysed for application-specific agent**	**Maximum LoC analysed for SQLite infrastructure-based agent**	**% of log entries retrieved by SQLite infrastructure-based agent**
WhatsApp	1,515,334	395,076	100
Telegram	1,025,467	-[*]	100
Signal	1,552,171	-[*]	100

[*] The same SQLite codebase is used to develop an SQLite infrastructure-based agent for all three apps. Therefore the lines of code is analysed only once.

application event metadata generated. Application event metadata generated by SQLite infrastructure-based logging agent translates roughly to the same metadata obtained by the application-based agents in the case of Telegram and Signal. With WhatsApp, however, application event log entries are missing the recipient value. Unlike Telegram and Signal, WhatsApp is closed-source. Therefore, parsing the generated logs was not as straightforward and possibly required further in-depth reverse engineering efforts to fully parse metadata within SQLite infrastructure-based log entries. That said, given that the log entries generated by the agent contain all the app events involving interactions with the database, additional reverse engineering effort can be carried out offline at a later stage to parse the generated logs.

In the case of application-specific JIT-MF drivers, evidence objects are identified and defined within a driver based on knowledge of the application codebase, which calls for a reverse engineering process. We use lines of code (LoC) as a metric to highlight the reverse engineering effort that needs to be made when developing application-specific and SQLite infrastructure-based logging agents. The second and third columns of Table 3 show the *maximum* LoC that need to be analysed using both approaches (application-specific and SQLite infrastructure-based logging agent) to select the relevant evidence object and thus develop a logging agent for each app. In the case of the application-specific logging agents, LoC for each app reflects the lines of code in the respective decompiled `java` source files. For the SQLite infrastructure-based agent, the LoC reflects the lines of code in `sqlite` source and header files. Since the same SQLite infrastructure-based agent was used for the three apps, the LoC value for the SQLite infrastructure-based agent is only considered once. Therefore, the codebase that needs to be reversed to develop an SQLite logging agent is much smaller than each of the applications' codebases and only needs to be reversed once since it is common for all three apps. The LoC values in the table portray the worst-case scenario, as reverse engineering efforts can be reduced further in the case of application-specific agents by using keyword searches (e.g. *Message*) – assuming an unobfuscated codebase – to narrow the search for the evidence object to a couple of classes. With infrastructure-based agents, public documentation of the infrastructure's interface can outline key exposed methods without going through the infrastructure codebase.

Threats to Validity. JIT-MF aims to ensure minimal forensic readiness by repackaging apps rather than requiring the device to be rooted. While minimal forensic readiness is a core property of JIT-MF, it falls outside this experiment's scope, as our aim is to assess the accuracy of a JIT-MF driver based on infrastructure rather than application-specific logic. Therefore, a rooted emulator was used to carry out the experiment, which enabled ease of automation and results-gathering, and drivers could be embedded directly on the device without app repackaging. While repackaging should not compromise the results obtained, this merits threats to validity as the results obtained assume that the driver can be installed.

5 DISCUSSION

While infrastructure-based forensic logging agents can significantly reduce the code comprehension effort required, selecting the proper infrastructure upon which a forensic logging agent is developed is critical to the agent's ability to produce the necessary evidence. SQLite, the storage management infrastructure employed by Android Architecture Components, was selected for our experiments. This infrastructure persists data from memory to a database on the user's phone. While our qualitative case study demonstrated the success of an SQLite infrastructure-based agent on three Android messaging apps, we expect that similar storage infrastructure usage is commonplace among the most popular apps, regardless of their category. Applications maintain their state by persisting essential app data and events to storage (either on the device or to cloud storage). Thus, applications are bound to use storage infrastructures that handle key application events even beyond messaging, potentially allowing such an infrastructure-based agent to be reused among multiple apps.

We assess the potential effectiveness of using an infrastructure-based approach to generate logging agents for Android apps at a larger scale by performing coverage analysis on the most downloaded Android apps from Google Playstore (550 apps in total, 44 of which fall in the messaging category). Using statistics from AppBrain (AppBrain API, 2022), Figure 4 shows that, potentially, developing two storage-related infrastructure-based logging agents would be enough to forensically enhance a large majority (94.7%) of the most popular apps. At

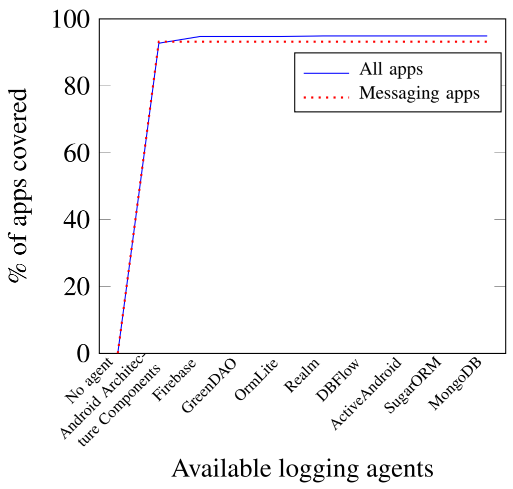

Figure 4: The number of storage infrastructure-based agents needed to forensically enhance a % of apps from the most downloaded 550 apps on Google PlayStore.

the same time, a single storage-related infrastructure-based logging agent could enhance 93.2% of messaging apps. This means a significantly small number of apps (5-7%) would require further code comprehension effort to develop the relevant logging agents.

Considerations & Limitations. Our coverage analysis indicates that storage-related infrastructure-based agents could significantly decrease the effort required when developing forensic logging agents for third-party apps by requiring fewer agents to enhance more apps successfully. This analysis, however, is based on statistics derived from AppBrain; thus, any limitations regarding how these statistics are generated can impact the results of our analysis.

While results from our case study show that less application-specific reverse engineering is required to develop an effective storage-related infrastructure-based agent, application-specific parsing is still necessary, to a lesser extent. Application evidence in memory is typically transformed (see Listing 3) when persisted to storage, regardless of the application category, requiring some form of application-specific parsing (see Listing 4). While the type of transformation is specific to the application and evidence, our case study showed that: i) this transformation is typically lossless (i.e. all the event metadata is retained in log entries) and structure-related, requiring some form of deserialisation; ii) the necessary steps to carve out all the metadata about key application events can quickly be determined upon observation.

This may not always be the case. Application developers may opt to transform evidence in memory so they can no longer be easily parsed, for instance, by persisting encrypted or compressed data to a local database. However, this may not be practical as is the case in the context of Android, for instance. File-based encryption (FBE) (Google, 2022) is enabled by default on devices running Android 10 or higher. Therefore, encrypting application data would lead to slower performance without any additional security benefit beyond the default.

6 RELATED WORK

Forensic Logging. Tracing system-level dependence is a popular technique for anomaly detection in desktop, server and mobile environments (Newsome and Song, 2005; King et al., 2005; King and Chen, 2003; Ammann et al., 2002; Yuan et al., 2017). Such techniques are typically used to record events (e.g. system calls, API calls) during execution so that causal dependencies between system subjects (e.g. process) and system objects (e.g., network socket or file) can be identified to detect malicious behaviour. Ultimately, traces of function/API calls comprise logs that serve an ulterior specific purpose, such as malware detection. In this work, we show that application events logged at the level of function/API tracing intended to provide deeper visibility are not inferior to enhanced forensic logging that benefits from application-specific logic knowledge.

Similarly to tools like BEEP (Lee et al., 2013), ProTracer (Ma et al., 2016), and WinLog (Ma et al., 2015), an infrastructure-based forensic logging agent, as proposed in this paper, calls for a proactive approach to forensically enhance application binaries for attack reconstruction. However, rather than relying solely on system and memory level dependencies, infrastructure-based agents may also rely on library and API calls. Calls to these dependencies are less complex to reverse engineer and can obtain semantically richer traces of logs obtained from lower-level dependencies (e.g. system calls).

Logging agents deployed by forensic enhancement tools complement tracing techniques by obtaining richer traces through custom tracing third-party applications. The logging entries (traces) generated can be used alongside other sources of evidence for analysis by endpoint visibility tools such as GRR Rapid Response and Velociraptor IR, provided that an appropriate parser is created for the traces.

Dynamic Instrumentation as an Enabler for Logging Agents. Instrumentation is not always necessary for log enhancement (Ma et al., 2015). However, in the case of logging agents for third-party applications it is the primary enabler (Toslali et al., 2021; Bellizzi et al., 2022). While the approach introduced in this paper was evaluated using an enhancement technique that leverages instrumentation, the main aim of our work involved developing logging agents that require less reverse engineering effort while mitigating the accuracy risks in the logs generated.

Android Forensic Enhancement. In the context of Android, DroidForensics (Yuan et al., 2017) enables multi-layer logging events on Android from different layers, while CopperDroid (Tam et al., 2015) proposes system-call logging. Rather than utilising a logging agent to log events of interest, these solutions require modifications to the underlying Android system. While this means that application reverse engineering and agents breaking compatibility are no longer an issue, users must significantly modify their devices.

Our work builds on previous research that introduces the JIT-MF framework (Bellizzi et al., 2020; Bellizzi et al., 2022; Bellizzi. et al., 2021), a forensic enhancement technique for Android that uses application-specific logging agents (drivers) to dump evidence from memory to aid investigations regarding benign application misuse. The cornerstone of the framework is a JIT-MF driver having application-specific properties related to the application-specific trigger points and objects that need to be dumped from memory. Previous work evaluated the impact black-box (non-application-specific) trigger points would have on the object (evidence) dumped (Bellizzi et al., 2020); however, the object itself was application-specific. Experiment results from this paper show that JIT-MF drivers should follow an infrastructure-centric approach, which reduces application reverse engineering effort by forgoing the need to work with application-specific objects and thus increasing the driver's stability across applications and versions.

7 CONCLUSION

This paper presents a new approach to developing post-deployment forensic logging agents, deployed during the forensic readiness stage of incident response, for third-party applications whose access to the source code is limited. This approach leverages interfaces to widely deployed infrastructure to minimise the reverse engineering efforts required to develop such agents, thereby increasing compatibility across applications and versions.

Our experiments have shown that SQLite can be leveraged in the context of Android messaging apps to develop infrastructure-based logging agents that remain prevalent across app versions and functional across three different applications. In doing so, we considerably reduce the reverse engineering effort while increasing compatibility across applications and their versions.

Coverage analysis based on data from AppBrain also shows that potentially, using this approach, 95% of most downloaded apps on Google Playstore can be forensically enhanced using at most two storage-related infrastructure-based forensic logging agents. This significantly reduces the number of agents that need to be developed and the individual apps that would need to be reverse-engineered. This result bodes well for JIT-MF and any other forensic logging enhancement technique aiming to provide deeper visibility through a third-party application-specific forensic logging agent.

ACKNOWLEDGEMENTS

This work is 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.

REFERENCES

Amer Owaida (2021). Wormable android malware spreads via whatsapp messages. https://www.welivesecurity.com/2021/01/26/wormable-android-malware-spreads-whatsapp-messages. Accessed: 9.11.2021.

Ammann, P., Jajodia, S., and Liu, P. (2002). Recovery from malicious transactions. *IEEE transactions on knowledge and data engineering*, 14(5):1167–1185.

AppBrain (2022). Appbrain: Monetize, advertise and analyze android apps. https://www.appbrain.com/stats/libraries. Accessed: 23.08.2022.

AppBrain API (2022). https://api.appbrain.com/v2/info/browse. Accessed: 20.10.2022.

Bellizzi, J., Vella, M., Colombo, C., and Hernandez-Castro, J. (2020). Real-time triggering of android memory dumps for stealthy attack investigation. In *NordSec*, pages 20–36.

Bellizzi., J., Vella., M., Colombo., C., and Hernandez-Castro., J. (2021). Responding to living-off-the-land tactics using just-in-time memory forensics (jit-mf) for android. In *SECRYPT*, pages 356–369.

Bellizzi, J., Vella, M., Colombo, C., and Hernandez-Castro, J. (2022). Responding to targeted stealthy attacks on android using timely-captured memory dumps. *IEEE Access*, 10:35172–35218.

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

Flubot (2021). Flubot malware – all you need to know & to act now. https://www.threatmark.com/flubot-banking-malware/. Accessed: 6.03.2021.

Google (2022). File-based encryption. https://source.android.com/security/encryption/file-based Accessed: 22.10.2022.

Jon Russell (2012). Stealth sms payment malware identified in chinese android app stores, 500,000 devices infected. https://thenextweb.com/news/stealth-sms-payment-malware-identified-chinese-app-stores-500000-android-devices-infected. Accessed: 2.10.2022.

King, S. T. and Chen, P. M. (2003). Backtracking intrusions. In *ACM SOSP*, pages 223–236.

King, S. T., Mao, Z. M., Lucchetti, D. G., and Chen, P. M. (2005). Enriching intrusion alerts through multi-host causality. In *NDSS*.

Lee, K. H., Zhang, X., and Xu, D. (2013). High accuracy attack provenance via binary-based execution partition. In *NDSS*, volume 16.

Ma, S., Lee, K. H., Kim, C. H., Rhee, J., Zhang, X., and Xu, D. (2015). Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC*, pages 401–410.

Ma, S., Zhang, X., Xu, D., et al. (2016). Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, volume 2, page 4.

Mace, J., Roelke, R., and Fonseca, R. (2015). Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393.

Newsome, J. and Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signature-generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4.

Octo (2022). Look out for octo's tentacles! a new on-device fraud android banking trojan with a rich legacy. https://threatfabric.com/blogs/octo-new-odf-banking-trojan.html. Accessed: 16.08.2022.

Tam, K., Fattori, A., Khan, S., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS Symposium 2015*, pages 1–15.

ThreatLandscape (2022). 2022 mobile threat landscape update. https://www.threatfabric.com/blogs/h1-2022-mobile-threat-landscape.html. Accessed: 6.03.2021.

Toslali, M., Ates, E., Ellis, A., Zhang, Z., Huye, D., Liu, L., Puterman, S., Coskun, A. K., and Sambasivan, R. R. (2021). Automating instrumentation choices for performance problems in distributed applications with VAIF. In *ACM SoCC*, pages 61–75.

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. (2012a). Be conservative: Enhancing failure diagnosis with proactive logging. In *USENIX OSDI*, pages 293–306.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2012b). Improving software diagnosability via log enhancement. *ACM TOCS*, 30(1):1–28.

Yuan, X., Setayeshfar, O., Yan, H., Panage, P., Wei, X., and Lee, K. H. (2017). Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging. In *ACM ASIACCS*, pages 666–677.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. (2017). Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *ACM SOSP*, pages 565–581.